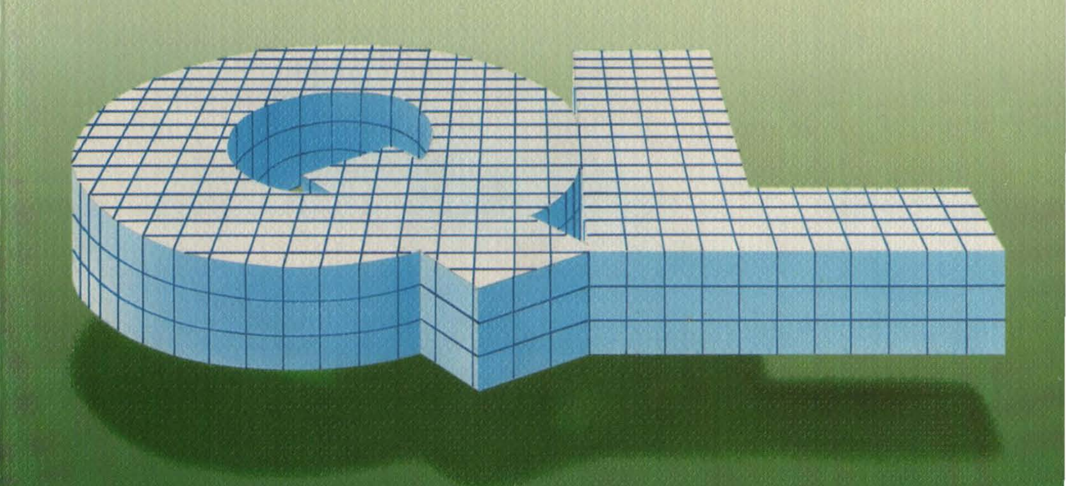
Boris Allan

Sinclair QL-Begleiter





Boris Allan · Sinclair QL-Begleiter

Boris Allan

Sinclair QL-Begleiter

Dr. Alfred Hüthig Verlag Heidelberg

Diejenigen Bezeichnungen von im Buch genannten Erzeugnissen, die zugleich eingetragene Warenzeichen sind, wurden nicht besonders kenntlich gemacht. Es kann also aus dem Fehlen der Markierung ® nicht geschlossen werden, daß die Bezeichnung ein freier Warenname ist. Ebensowenig ist zu entnehmen, ob Patente oder Gebrauchsmusterschutz vorliegen.

Übersetzt von
Dietrich Alfred Schilling
Fliederweg 15
6115 Münster

Die englische Originalausgabe ist erschienen unter dem Titel:
Boris Allan, THE SINCLAIR QL COMPANION bei Pitman Publishing
Limited London

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Allan, Boris:

Sinclair-QL-Begleiter / Boris Allan. [Übers. von
Dietrich Alfred Schilling]. – Heidelberg : Hüthig,
1985.
ISBN 3-7785-1101-7

Vorwort (des Verfassers)

C. H. Lindsey und S. G. van der Meulen schrieben (zu Informelle Einführung in ALGOL 68): „Da ALGOL 68 eine sehr rekursiv strukturierte Sprache ist, kann sie kaum erklärt werden, solange sie nicht erklärt ist." Dann beginnen sie und versuchen, den Leser schrittweise zu führen, „ohne die Denkprozesse zu einem rekursiven Knoten zu verschlingen." Diese Autoren haben meine Sympathie.

Meine Probleme sind nicht so extrem wie die von Lindsey und van der Meulen, denn SuperBASIC kann auf dem QL benutzt werden wie „normales" BASIC (was auch immer Sie „normales" BASIC nennen). Doch es wäre Verschwendung, SuperBASIC nur als normales BASIC zu verwenden, da zuviele Eigenschaften verloren gingen.

Unter Berücksichtigung der Grenzen, die der Sprache durch die Implementierung in einer interaktiven Umgebung auf einem Mikrocomputer auferlegt sind, hat SuperBASIC in seiner Philosophie viel Gemeinsames mit der Programmiersprache ALGOL 68. Will man SuperBASIC voll nutzen, muß man es als rekursiv strukturierte Sprache sehen, mit klar abgegrenzten Elementen, die der Vorstellung der „closed clauses" von ALGOL 68 sehr ähnlich sind.

Meine Untersuchung von SuperBASIC war absichtlich ganz anders, als sie es bei einem gewöhnlichen BASIC gewesen wäre; so benutze ich zum Beispiel nach den ersten Programmen keine Zeilennummern mehr. Ich kann mich auch nicht mit allen Einzelheiten von SuperBASIC beschäftigen, denn das hieße das Unmögliche zu versuchen, aber ich habe versucht, dem Leser ein Gefühl für die Ideen hinter der Sprache zu geben. Der beste Weg, dieses Gefühl für eine Sprache zu bekommen ist deren Benutzung, also das Programmieren in der Sprache. Deshalb habe ich viele Beispiele für die m. E. wichtigsten Programmierthemen und nützlichen Anwendungen ausgearbeitet.

Der QL ist anders als die anderen, da er der erste erschwingliche Mikrocomputer mit einem 16-Bit-Mikroprozessor ist. Dieser Aspekt ist so wichtig, daß ich einer detaillierten Untersuchung des QL Motorola MC 68008 Mikroprozessors viel Zeit und Platz gewidmet habe (und wenig einem anderen Chip, dem Intel 8049 Mikroprozessor). Da die Sprache in einer Box als Teil der Hardware vorliegt, habe ich auch die Hardware-Charakteristika kurz beschrieben.

Meine „Beschimpfungen“ gehen an Robert und Alfred für ihre Sticheleien, meine Danksagungen an meine Familie für ihre Geduld (immerhin spricht sie noch mit mir). Zu guter Letzt, besonderen Dank an A. und G.

Inhalt

1. Einführung zum QL

Der QL ist nicht nur in zwei wichtigen Punkten außergewöhnlich, er hat auch einige weniger wichtige, ausgezeichnete Details.

Das vielleicht wichtigste Merkmal des QL ist seine Programmiersprache SuperBASIC, die so intelligent konzipiert ist, daß sie viele andere BASIC-Versionen merkwürdig beengend aussehen läßt.
SuperBASIC ist in seiner Konzeption so anders, daß die in SuperBASIC geschriebenen Programme gar nicht wie BASIC aussehen. Die Bedeutung von SuperBASIC geht über den QL hinaus und die dahinter stehenden Ideen verdienen eine viel größere Beachtung.

Der andere bedeutende Aspekt, der vielleicht nicht ganz so wichtig wie SuperBASIC ist, besteht in der Verwendung des Motorola MC 68008 Mikroprozessors.
Der MC 68008 ist ein sehr schneller und gut durchdachter Chip, verfügt über einen vorzüglichen und starken Befehlsvorrat und ist unbestreitbar der beste der zur Zeit vorhandenen Mikroprozessoren mit einem 8-Bit Datenbus.

Übersicht

Es gibt deshalb zwei Hauptthemen in diesem Buch. Ein Aspekt ist die ausführliche Beschreibung, wie SuperBASIC arbeitet, sowie Vergleiche zu anderen BASIC-Versionen und (noch wichtiger), Vergleiche mit anderen Computer-Programmiersprachen. Bei der Beschreibung von SuperBASIC wurde der Erläuterung und Verdeutlichung der Struktur größere Bedeutung zugemessen als den Details der Operationen, die im Handbuch nachgelesen werden können.

Diese Wertung ist zum Teil auf die Politik der Sinclair Forschung, stetige Entwicklung, zurückzuführen. Wenn sich auch die Struktur kaum ändern wird,

so sind doch Änderungen in kleinen Details unvermeidbar, wenn Widersprüche entdeckt werden. Das Hauptziel dieses Buches ist der Nachweis, daß SuperBASIC eigentlich kein BASIC ist. Nicht wegen der Bedeutung der Namen, vielmehr wegen der Behinderung, die sich bei der Benutzung der neuen Sprache unter Beibehaltung der alten Ideen und Gewohnheiten des „alten“ BASIC ergeben.
Das ist ein Grund, warum nach den ersten Beispielen die Zeilennummern absichtlich nicht mehr verwendet werden. Wenn man ein SuperBASIC-Programm ohne Zeilennummern und ohne LET-Befehl betrachtet, dann stellt man eine große Ähnlichkeit mit der Programmiersprache PASCAL fest. Tatsächlich kann mit ruhigem Gewissen gesagt werden, daß SuperBASIC stärker als PASCAL ist. Der andere wichtige Aspekt ist der MC 68008 Chip und der etwas weniger aufsehenerregende Intel 8049 Einchip-Computer. Beide werden ausführlich beschrieben.

Dieses Vorgehen resultiert zum einen aus der Überzeugung, daß es für das Verständnis des QL wichtig ist zu wissen, wie diese Mikroprozessoren arbeiten, zum anderen aus dem Glauben, daß diese Informationen in sich interessant sind. Außerdem wird davon ausgegangen, daß einige Besitzer des QL diesen in der Maschinensprache programmieren wollen.

Grafik

Die vom Hersteller gelieferten Software Pakete zum QL werden hier nicht behandelt. Die Beschreibung und Programmdokumentation dafür ist gut und ausreichend. Außerdem handelt es sich um zweckorientierte Software, deren ausführliche Beschreibung keine Hilfe für das Verständnis der Programmiertechnik dieses Mikrocomputers gibt.
Statt dessen wird hier ein Satz getesteter Routinen für ein Turtle-Grafik-System für den QL geliefert, der leicht nach den persönlichen Anforderungen modifiziert werden kann. Die Grafik wird gegenüber dem Sound mehr betont, da dabei die Möglichkeiten des Systems besser demonstriert werden können; außerdem entspricht es mehr meinen eigenen Prioritäten.

Ich finde Computergrafik so interessant und befriedigend, daß ich einfach unterstelle, daß andere genauso verzaubert werden. Da ich absolut unmusikalisch bin, kann ich mich für Computermusik einfach nicht begeistern.
Selbst eine flüchtige Betrachtung der Turtle-Grafik-Routinen zeigt die Stärke von SuperBASIC und seine leichte Anwendbarkeit.

Gedanken zum QL-Begleiter

Der Begleiter soll weder ein Hardware- noch ein Software-Handbuch sein; vielmehr soll er Hilfestellung geben, damit Sie mit Ihrem QL „zusammenkommen". Er soll Ihnen helfen zu verstehen, warum man einige Sachen besser so macht als anders.

Da die Probleme mit Mikroantrieben, seriellen Ports und ROM-Kassetten ziemlich „haarig" sein können und obendrein mit jeder neuen Version wechseln, liegt beim Begleiter die Betonung auf den konstanten Elementen des QL. Eine Konstante ist die Philosophie und das Ziel von SuperBASIC, eine andere ist die Architektur und der Befehlssatz des MC 68008. Eine weitere ist die Arbeitsweise der Turtle-Grafik. Das sind meine Konstanten.

Zweifellos werden Bücher zu dieser, jener und anderen Besonderheiten des QL erscheinen. Ich habe versucht, dem Leser darzustellen, warum bzw. warum nicht, der QL ein Riesensprung vorwärts ist.

2. Wir bauen BASIC-Strukturen

Der Computer ist die wandlungsfähigste Maschine, die je erfunden wurde. Seine Fähigkeit, etwas anderes vorzutäuschen, macht ihn zum Chamäleon unter den Maschinen. Ein Roboter (eine Variante des Computers) kann ohne weiteres eine Person imitieren, die eine komplizierte Maschine bedient.

Seymour Papert schrieb in seinem Buch MINDSTORMS: „Der Computer ist der Proteus unter den Maschinen. Sein Wesen ist seine Vielseitigkeit; seine Fähigkeit zu simulieren. Da er in tausend verschiedenen Formen kommen und tausend Funktionen ausführen kann, gibt es ihn für jeden Geschmack." In der griechischen Mythologie war Proteus der weise alte Mann der See, der sich vor neugierigen Fragern versteckte, indem er sich verwandelte. Wer den listigen alten Mann fangen und halten konnte, bekam seine Zukunft vorausgesagt.

Vielleicht dachte Seymour Papert nur an Proteus' Fähigkeit sich zu verwandeln, aber ein besseres Verständnis der Computer könnte uns wirklich helfen zu erkennen, was in Zukunft auf uns zukommt. Die Evolution der Computer geht schnell vor sich. Es scheint aber, als ob die Entwicklung der Programmiersprachen hinterherhinkt.

Programmiersprachen

Im Wesentlichen ist die Proteus'sche Natur des Computers (seine Flexibilität und seine Möglichkeiten) auf die menschliche Fähigkeit zurückzuführen, ihn zu programmieren. Der Computer ist in der Regel nicht besser als das Programm, das er im Moment ausführt.

Es gibt viele Mikrocomputer mit phantastischer Geschwindigkeit und Speichergröße, die eine so primitive BASIC-Version benutzen, daß sie in ihrer Ausdrucksstärke dem Trompeten eines Elefanten gleicht. Die Anforderun-

gen an die Computer wachsen und wechseln schnell; der QL hat dieses gerade bewiesen. Der im QL benutzte Motorola MC 68008 Mikroprozessor ist wesentlich stärker als die in vielen anderen (oft wesentlich teureren) kommerziellen Mikrocomputern. Ein weniger starker Mikroprozessor ist zum Beispiel der Intel i8088, wie er im IBM PC eingesetzt ist. So leistungsfähig der MC 68008 im Verhältnis zu den meisten Mikroprozessoren auch ist, so darf nicht vergessen werden, daß der Programmierer ein Mensch ist und Menschen (wie auch Maschinen) nicht unfehlbar sind (Perfektion gibt es nur in unseren Träumen und im Himmel).

Der QL benutzt eine Variante der Programmiersprache BASIC. Ganz genau genommen ist Sinclairs SuperBASIC (das ist der richtige Name) eigentlich keine BASIC-Version, aber eine derartige Argumentation ist sinnlos. Es gibt ausreichend Ähnlichkeiten zwischen anderen BASIC-Versionen und SuperBASIC, so daß man SuperBASIC mit ruhigem Gewissen als BASIC-Dialekt bezeichnen kann.

Es soll hier auch darauf hingewiesen werden, daß vielfach eine Standardisierung von BASIC angestrebt wird (einer dieser Versuche ist MSX BASIC, das von vielen japanischen Herstellern unterstützt wird). Alle derartigen Versuche sind genauso zum Scheitern verurteilt wie die Bestrebungen der französischen Regierung, die französische Sprache durch Gesetze und Verordnungen von Fremdworten zu befreien.
Alle Anstrengungen zur Standardisierung sind vergeblich, da die Computer von Menschen benutzt und von Menschen hergestellt werden. Von Menschen, die ihre Produkte mit der bestmöglichen Software für alle möglichen Zwecke verkaufen wollen. Standardisierung funktioniert nur in akademischen Kreisen (z. B. PASCAL) oder in kommerziellen, wo das Budget des US Verteidigungsministeriums diese Wirkung erzielt (z. B. COBOL). Für viele derartige Anwendungen ist eine Standardisierung sinnvoll.

In jeder BASIC-Version (oder jeder anderen Programmiersprache) findet man bestimmte Schlüsselfunktionen:

- Input, die Eingabe von Information in den Computer;
- Output, die Ausgabe der Information durch den Computer;
- außerdem Berechnungen und Vergleiche.

Bei der Untersuchung einer Sprache ist außerdem die Frage wichtig, auf welche Arten die Reihenfolge der Befehle eines Programms, in Abhängigkeit von den Umständen, variiert werden kann.

Wir benötigen nur drei Programme zur Darstellung dieser Aspekte. Das erste dieser Programme ist leicht zu erkennen und könnte problemlos auch auf dem Spectrum, dem Commodore 64 oder irgendeinem der tausend anderen Mikrocomputer laufen. Die beiden anderen, wesentlich eleganteren Programme benutzen das offensichtlich effizientere SuperBASIC.

Programm-Beispiel 1: Zahlenraten in BASIC

Als erstes müssen wir das Problem verstehen. Wir wollen ein einfaches Programm zum Zahlenraten schreiben, das vielleicht sogar denen helfen kann, die in Arithmetik nicht besonders gut sind. Es ist eines der einfachen Spiele, bei dem der Lehrer (Computer) eine Nummer zwischen 0 und 100 wählt und der Schüler (Computer-Benutzer) die Zahl nicht genannt bekommt, sondern sie erarbeiten muß.
Der Benutzer rät eine Zahl und bekommt jeweils gesagt, ob diese Zahl höher oder niedriger als das Original (die vom Computer ausgewählte Zahl) ist.

Es ist leicht zu erkennen, daß selbst ein derartig einfacher Vorgang viele verschiedene Zahlenfolgen hervorrufen kann. Eines der Probleme besteht darin, daß das Programm praktisch alle möglichen Umstände berücksichtigen muß. Das Spiel taugt z. B. nichts, wenn die vorgegebene Nummer bei jedem Programmstart die gleiche ist.

Die erste Aufgabe besteht also darin, unter Berücksichtigung der Spielbedingungen einen Weg auszuarbeiten, der alle – zumindest jedoch die vorhersehbaren – Eventualitäten abdeckt und lösen kann. In der Programmierung nennt man die Methode, mit der ein Problem gelöst wird, einen ALGORITHMUS (ein Wort mit dem gleichen Ursprung wie ALGEBRA).

Die Reihenfolge der Schritte könnte wie folgt sein:
1. Schritt: Wähle die zu erratende Nummer;
2. Schritt: Setze die Anzahl der geratenen Zahlen auf 1;
3. Schritt: Frage nach der zu ratenden Zahl;
4. Schritt: Wenn die Antwort richtig ist, gehe zum Schritt 8;
5. Schritt: Information ausgeben, ob die erratene Zahl größer oder kleiner als die vorgegebene ist;
6. Schritt: Erhöhe die Anzahl der geratenen Zahlen um 1;
7. Schritt: Gehe zu Schritt 3;
8. Schritt: Programm-Ende Information ausgeben;
9. Schritt: Ende.

Beachte die verschiedenen Bedeutungen in diesem Algorithmus

Input:	Das ist die jeweils geratene Zahl;
Output:	Die Information, wie die geratene Zahl zu der vorgegebenen liegt (größer/kleiner);
Kalkulation:	Welche Zahl soll vom Computer vorgegeben werden;
Vergleich:	Wie ist das Verhältnis zwischen Vorgabe und geratener Zahl;
Verzweigung:	Wie soll das Programm entsprechend der geratenen Zahl weiterarbeiten.

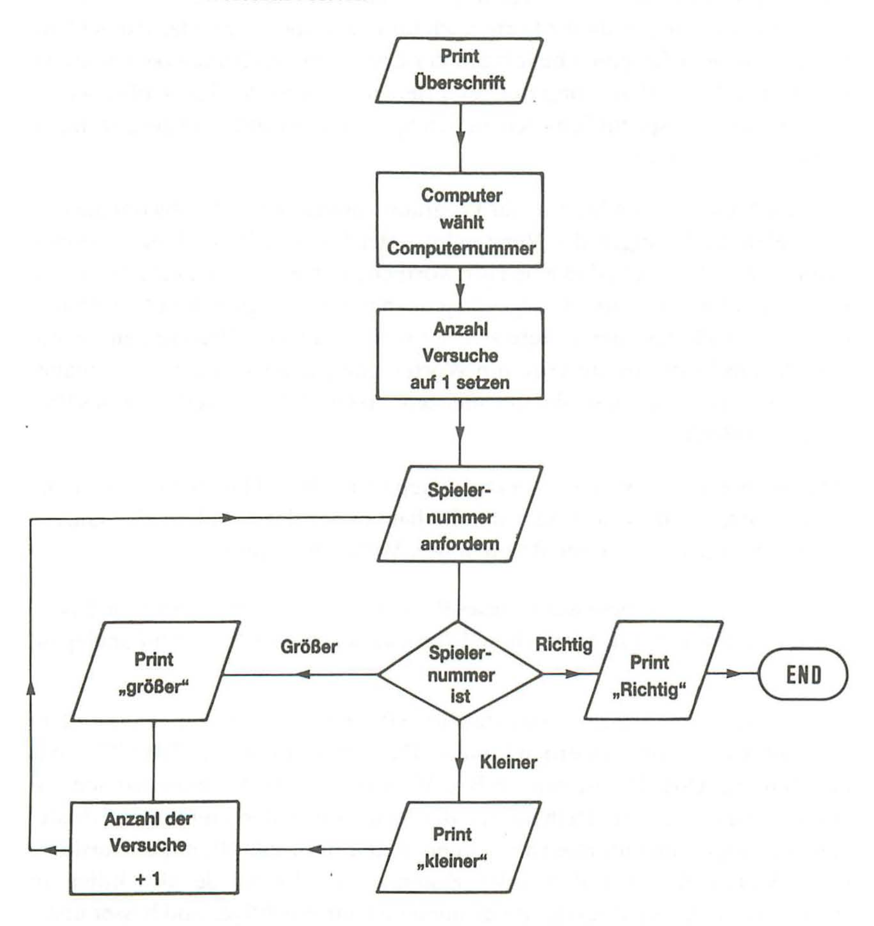


Abb. 2.1 Flußdiagramm zum Zahlenrateprogramm

Der Algorithmus ist als Programmablaufplan (Flußdiagramm) symbolisch in Abbildung 2.1 dargestellt. Ein- und Ausgabe werden dabei durch ein schräges Viereck stilisiert, Berechnungen durch ein Rechteck und Vergleiche sowie Entscheidungen durch einen Rhombus. Die Linien mit Pfeil zeigen den Weg von einem Symbol zum anderen. Dieser Algorithmus kann leicht in ein BASIC-Programm übersetzt werden (beachte: Es ist noch kein SuperBASIC-Programm).

Das Programm (Abb. 2.2) ist recht interessant. Wir stellen zum Beispiel fest, daß das Programm in dieser Form auch für einen Spectrum oder einen Commodore 64 oder für einen beliebigen der ersten (in den frühen 60er Jahren), mit Basic arbeitenden Computer geschrieben sein könnte. Tatsächlich wurde das Programm ursprünglich auch für den Spectrum verfaßt, und die Spectrum-Version ist fast identisch.

Für den Nichteingeweihten ist das Programm unverständlich, obwohl die vorstehenden Erklärungen das Verständnis erleichtern sollten. Ganz allgemein kann man wohl sagen, daß eine gute wörtliche Erklärung eine gute Hilfe sein kann, ein Flußdiagramm ist dagegen jedoch nur von begrenztem Nutzen und eigentlich auch nur eine weitere zu erlernende Sprache, allerdings eine Sprache, die aus Symbolen anstelle von Worten und Zahlen besteht. Ein genaues Flußdiagramm kann man ohnehin meistens erst nach Fertigstellung eines Programms anfertigen.

Bedenken wir, wie wir das Problem angepackt haben. Haben wir uns auf die Einzelheiten konzentriert oder durchschauten wir das Problem als Ganzes? Haben wir nicht vor lauter Bäumen den Wald übersehen?

Es lohnt sich, die Arbeitsweise dieses Programms zu erklären, denn die Erklärung zeigt deutlich auf, wieso diese Konstruktion ziemlich roh und unelegant ist.

Ich habe schon mehrfach betont, daß diese Programmart auch für viele andere Computer geschrieben werden könnte. Deshalb ist es nur ein BASIC-, aber kein SuperBASIC-Programm. In BASIC beginnen alle Programmzeilen mit einer Nummer, die die Reihenfolge der Programmzeilen enthält. Im vorstehenden Programm enthalten alle Zeilen, außer der Zeile 60, nur ein ausführbares Statement (= Befehl oder Kommentar). Es wurde absichtlich so codiert, denn die Struktur des Programmes ist durchsichtiger und besser überschaubar.

```
 10 REM  HILO, Ein Zahlenratespiel
 20 REM
 30 REM  Version für den QL in BASIC, Autor Boris Allan
 40 REM
 50 PRINT  "HILO- Raten Sie eine Zahl zwischen 0 und 100,
             und versuchen Sie, meine Vorgabe zu finden"
 55 REM
 60 RANDOMISE :  LET computernummer = RND (0 TO 100)
 65 REM
 70 LET  versuche = 1
 75 REM
 80 INPUT  "Welche Zahl raten Sie" ; nummer
 85 REM
 90 IF     nummer > computernummer THEN PRINT  "Ihre Zahl
           ist zu gross" : REM  Erster Vergleich
 95 REM
100 IF  nummer < computernummer THEN PRINT  "Ihre Zahl
        ist zu klein" : REM  Zweiter Vergleich
105 REM
110 IF  nummer=computernummer THEN GOTO 140 : REM
        letzter Vergleich
115 REM
120 LET  versuche=versuche+1
125 REM
130 GOTO 80:REM Zurück zur Eingabe
135 REM
140 PRINT  "Richtig nach" ; versuche; "Versuchen" : REM
           Letzte Zeile
145 REM
150 STOP
```

Abb. 2.2 Das codierte Programm, BASIC-Version

Es gibt noch eine Reihe anderer Programmiersprachen, bei denen Zeilennummern verwendet werden. Immerhin wird dadurch die Programmpflege und -änderung erleichtert, denn anhand der Nummer lassen sich Zeilen bequem löschen oder einfügen. Das gewöhnliche BASIC ist jedoch eine der wenigen Sprachen, die die Zeilennummern auch als symbolische Adresse für die Programmkontrolle verwendet.

Alle mit REM beginnenden Zeilen sind Kommentare (= REMarks); das heißt Informationen für den Autor, die vom Computer ignoriert werden. Der erste ausführbare Befehl steht demzufolge in der Zeile 50. Er gibt eine Überschrift auf dem Bildschirm aus, die dem Benutzer sagt, was er zu erwarten und zu tun hat.

Zeile 60 enthält, durch einen Doppelpunkt getrennt, zwei ausführbare Anweisungen. Die erste (RANDOMISE) initialisiert den Zufallsgenerator, der dann eine zufällige Zahl zwischen 0 und 100 generiert. Die RND-Funktion gibt es in fast allen BASIC-Dialekten, die Art der Bereichsangabe ist eine Besonderheit von SuperBASIC.

Zeile 70 setzt den Zähler, der angibt, wie oft geraten wird, auf 1; und in Zeile 80 wird der Spieler nach seiner Zahl gefragt: „Welche Zahl raten Sie?"

Gleichzeitig erwartet das Programm die Eingabe der Zahl über die Tastatur. Zeile 90 vergleicht die Eingabe mit der „computernummer". Wenn (= IF) die geratene Zahl (= nummer) größer als die Vorgabe des Computers ist, dann (= THEN) soll auf dem Bildschirm die Information „Ihre Zahl ist zu groß" ausgegeben werden. In der nächsten ausführbaren Anweisung (Zeile 100) wird der entgegengesetzte Fall überprüft.

In Zeile 110 ist vorgegeben, was geschehen soll, wenn die geratene Zahl richtig ist. Wenn (= IF) die eingegebene Zahl identisch mit der vom Computer vorgegebenen Zahl ist, dann (= THEN) soll das Programm aus der Befehlsfolge ausbrechen, zur Zeile 140 (GOTO 140) gehen und dort weiterarbeiten. Das Programm überspringt also die Zeilen 115 bis 135. Bei Ausführung des Befehls in Zeile 140 wird der Benutzer über den Bildschirm informiert: „Richtig nach"; versuche; (= Anzahl der Versuche) „Versuchen". Das Programm endet mit der STOP-Anweisung in Zeile 150.
Falls die geratene Zahl nicht mit der „computernummer" identisch ist, erfolgt kein Sprung zur Zeile 140, sondern die Programmkontrolle geht auf die nächste ausführbare Anweisung nach Zeile 110 über. Da Zeile 115 ein REMark

(= Kommentarzeile) ist, wird bei Zeile 120 der Inhalt der Variablen „versuche“ um 1 erhöht.

An dieser Stelle muß betont werden, daß viele Computer mit primitiveren BASIC-Dialekten arbeiten, die Namen von Variablen in dieser Länge (computernummer) nicht verkraften (zum Beispiel das BASIC des APPLE II oder das der Commodore-Maschinen oder auch das Original Dartmouth BASIC). Die Variable „versuche“ wird vom Commodore 64 als „VE“ interpretiert, da in seiner BASIC-Version nur 2 Buchstaben/Zeichen für Variablen vorgesehen sind; auch den Unterschied zwischen Groß- und Kleinbuchstaben gibt es nicht.

Die Verzweigung, der Sprung von Zeile 110 nach Zeile 140, ist ein „bedingter Sprung“. Das heißt, er wird nur ausgeführt, wenn die Bedingung (nummer = computernummer) erfüllt ist. Der Sprung von Zeile 130 dagegen ist „unbedingt“. Wenn diese Zeile erreicht wird, dann wird immer zur angegebenen Zeilennummer verzweigt. In diesem Fall zur Zeile 80. Es ist offensichtlich, daß die Zeile 135 zum Beispiel nie angesteuert wird: entweder springt das Programm zu einer der davorliegenden (Zeile 80) oder das Programm verzweigt zu einer dahinterliegenden Zeile (Zeile 140).
Es geschieht also nichts weiter, als daß das Programm zwischen den Zeilen 80 und 130 in einer „Schleife“ arbeitet, bis die geratene Zahl mit der Vorgabe des Programms übereinstimmt. Um es ganz deutlich zu sagen: der Benutzer muß solange raten, bis er die richtige Zahl gefunden hat.

Programm-Beispiel 2: Zahlenraten in SuperBASIC, mit Iteration

Das Programm-Beispiel (Abbildung 2.2) war, zumindestens annähernd, auf dem Flußdiagramm (Abb. 2.1) aufgebaut. Wir wollen jetzt versuchen, die wirkliche Struktur des Programms zu finden, die im Flußdiagramm (Abb. 2.1) durch die intensive Beschäftigung mit den Einzelheiten nicht deutlich hervortrat. Im letzten Satz des vorigen Abschnitts wurde ganz klar gesagt: Rate eine Zahl, bis die richtige Zahl gefunden ist.

Abbildung 2.3 zeigt ein Diagramm, in dem versucht wird, sich auf das Wesentliche des Problems zu konzentrieren. Man beachte, daß es hier zwei Hauptkontrollmechanismen gibt: der eine wurde „FALSCH?“ genannt und der andere „IMMER“. Diese zwei Mechanismen ergeben die Basis jeglicher Kontrolle in allen Programmiersprachen.

Der erste (also: FALSCH?) kann wahrscheinlich am leichtesten als das Ergebnis der IF-Bedingung erkannt werden: der FALSCH-Mechanismus entspricht dem bedingten Sprung im HILO BASIC-Programm.

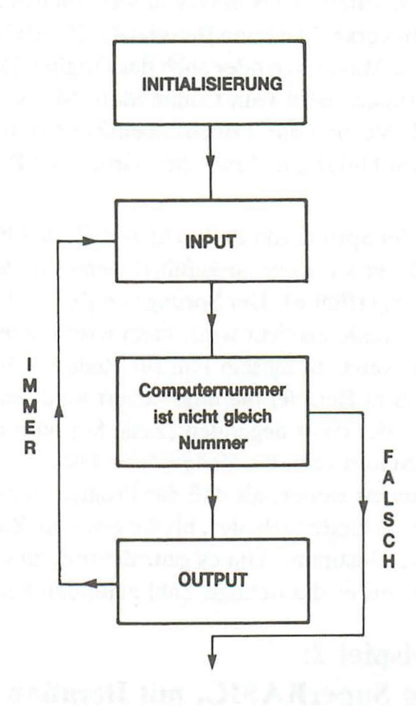


Abb. 2.3 Ein Kontroll-Diagramm zum HILO-Programm

Der zweite Mechanismus (also IMMER) entspricht der unbedingten Verzweigung in Zeile 130 (Abb. 2.2). Immer wenn dieser Befehl im Programmablauf erreicht wird, erfolgt der Sprung.

Der unbedingte Sprung ist aber nicht der einzige Weg zur Erreichung des IMMER-Weges, denn wenn es eine Möglichkeit zur Programmierung von Schleifen gibt, dann sollte sie auch genutzt werden. Im SuperBASIC gibt es die REPeat-Logik (Repeat = Wiederhole), die Programmschleifen eingrenzt und übersichtlich macht. Mit dieser Konstruktion werden die Befehle zwischen REPeat und END REPeat solange ausgeführt, bis die vorgegebene EXIT-Bedingung eintritt.

Das Format der REPeat-Logik ist wie folgt:

```
REPeat      schleifenname
            ausführbare Befehle 1
            bedingter EXIT schleifenname
            ausführbare Befehle 2
END REPeat schleifenname
REM  der  EXIT  führt zu diesem Statement.
```

Diese Logik kann in einem der Abbildung 2.3 sehr ähnlichen Diagramm dargestellt werden.

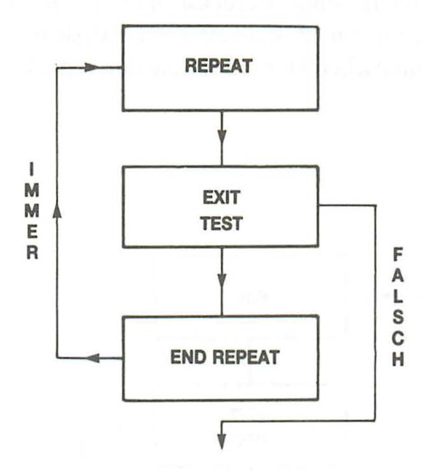


Abb. 2.4 Das REPeat-Kontroll-Diagramm

Der Gebrauch der REPeat-Logik ist ein typisches Beispiel für eine Iteration, d. h. eine Aktion oder eine Folge von Aktionen wird immer wieder durchgeführt, ob eine Ende-Bedingung definiert wurde oder nicht. Eine andere Form der Iteration ist die FOR-Schleife, die, im Gegensatz zur u. U. endlosen Wiederholung der REPeat-Schleife, nur eine bestimmte, vorgegebene Anzahl von Durchläufen kontrolliert. Auch hier besteht die Möglichkeit des bedingten Ausbruchs aus der Schleife.

Das Format der FOR-Schleife sieht so aus:

```
FOR variable  =  Liste der Werte
    ausführbare Befehle 1
    bedingter  EXIT  variable
    ausführbare Befehle 2
    evtl  NEXT  variable
    ausführbare Befehle 3
END FOR  variable
REM  der  EXIT  landet hier, die „normal" beendete Schleife
                                                 ebenfalls.
```

Wenn man die FOR-Logik, ohne Berücksichtigung von NEXT (das dem END FOR entspricht), untersucht, stellt man fest, daß sie mit zwei FALSCH-Wegen und einer automatischen Heraufzählung durch die FOR-Liste ausgestattet ist.

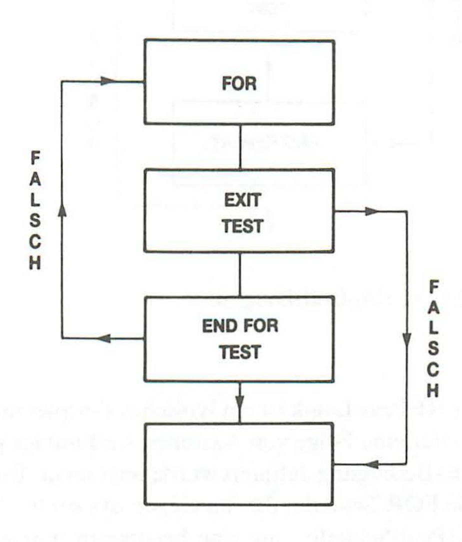


Abb. 2.5 Das FOR Kontroll-Diagramm)

Die besondere Konstruktion der FOR-Liste macht sie enorm flexibel: sie kann im Extremfall sogar aus Einzelwerten (Konstanten) bestehen:

```
FOR  variable = 1, 5, 6, 12, 29.
```

Bei diesem Beispiel werden der Variablen nacheinander die 5 vorgegebenen Konstanten zugewiesen.
Die beiden nächsten Beispiele entsprechen der konventionellen FOR-Schleife:

```
FOR variable  =  1  TO  20
FOR variable  =  1  TO  20  Step 2
```

Die erste FOR-Schleife wird (implizit) in Einer-Schritten durchlaufen; die zweite FOR-Schleife arbeitet, entsprechend der Angabe, in Zweier-Schritten.
Alle diese Formate können im SuperBASIC kombiniert werden:

```
FOR variable  =  1 TO 20, 29, 30  TO  40  STEP 2
```

Und in ihrem speziellen, gekürzten einzeiligen Format ist die FOR-Schleife besonders wirkungsvoll. Als Beispiel die Ausgabe der Quadrate der Zahlen von 1 bis 10:

```
FOR variable  =  1 TO 10:  PRINT variable * variable
```

Die Anweisung NEXT oder END FOR wird hier nicht benötigt.

Ein bedeutendes Charakteristikum von SuperBASIC ist die Art, wie der Umfang der REPeat- und FOR-Schleifen begrenzt wird. Diese umfangreiche und trotzdem klare Begrenzung dieser Schleifen kann dem Programmierer viele der Irrtümer und Fehler ersparen, für die andere BASIC-Dialekte so anfällig sind. In den meisten BASIC-Versionen wird dem NEXT kein Schleifen-Bezeichner beigefügt, der dem FOR entspricht. Ein Grund, warum viele Handbücher so schlecht zu lesen sind, liegt m.E. darin, daß dem NEXT-Befehl oftmals kein Schleifenzähler folgt. Wenn solche wichtigen Kleinigkeiten ignoriert werden, dann ist die ganze Sache schlecht durchdacht.

```
10 REM  HILO, Ein Zahlenratespiel
20 REM
30 REM Diese Version für den QL in iterativer Form
35 REM   ist vom Autor  Boris Allan
40 REM
50 PRINT "HILO - Raten Sie eine Zahl zwischen 0 und 100
         und versuchen Sie, meine Vorgabe zu finden"
60 RANDOMISE : LET computernummer = RND (0 to 100)
65 REM
70 LET versuche = 1
75 REM
80 REPeat  spiel
85 REM
90 INPUT "Welche Zahl raten Sie " ; nummer
95 REM
100 IF  nummer = computernummer  THEN  EXIT spiel
105 REM
110 IF  nummer   computernummer  THEN
115 REM
120 PRINT "Ihre Zahl ist zu gross"
125 REM
130 ELSE
135 REM
140 PRINT "Ihre Zahl ist zu klein"
145 REM
150 END IF
155 REM
160 LET  versuche = versuche + 1
165 REM
170 END REPeat  spiel
175 REM
180 PRINT "Richtig nach " ; versuche ; " Versuchen"
185 REM
190 STOP
```

Abb. 2.6 HILO-Programm SuperBASIC in der iterativen Form

Betrachten wir jetzt das Programm in Abb. 2.6. Nach einer kurzen Initialisierungsphase, in der die „computernummer" gewählt und der Zähler „versuche" auf 1 gesetzt wird, erfolgt die Wiederholung des Spiels in der Schleife REPEAT SPIEL. Das Spiel selber besteht aus der Eingabe einer Zahl durch den Spieler, und wenn diese Zahl richtig ist, erfolgt der Aussprung (EXIT) aus der REPeat-Schleife zu der Anweisung hinter END REPeat spiel. Die Kontrolle auf das Ende der Schleife erfolgt in einer einzeiligen, bedingten Anweisung, wie sie auch im vorigen Programm benutzt wurde.

Wenn der EXIT nicht ausgeführt wird, dann ist die eingegebene Zahl nicht richtig. Jetzt wird geprüft, ob „nummer", also die gespeicherte Eingabe, größer als die „computernummer" ist (IF nummer > computernummer). Wenn ja (THEN), dann wird „Ihre Zahl ist zu groß" am Bildschirm angezeigt. Andernfalls (ELSE) lautet die Ausgabe auf dem Bildschirm „Ihre Zahl ist zu klein". Diese neue Form der IF-Anweisung, mit END IF zur Begrenzung, hat große Ähnlichkeit mit anderen Programmiersprachen. Sie ist z. B. erheblich stärker als die IF... THEN... ELSE Logik im BBC-BASIC.
Nachdem dann der Zähler „versuche" um 1 erhöht wird, beginnt die REPeat-Schleife von vorn.

Bei den Erläuterungen dieser neuen Programmversion werden die Zeilennummern nicht benötigt, da sie lediglich als Zeilennummern, nicht aber als Programmadresse dienen. Das Programm ist trotzdem gut verständlich, da die Referenzadressen durch eindeutige Namen bezeichnet sind. So ist zum Beispiel die Begrenzung der Programmschleife „spiel" durch REPeat spiel bis END REPeat spiel eindeutig und übersichtlich, ähnlich den Klammern bei mathematischen Formeln und Berechnungen.

Der Bezeichner „spiel" ist also ein Etikett, das nicht nur den Umfang der REPeat-Schleife begrenzt, sondern auch das SuperBASIC-System informiert, zu welchem Punkt jeweils verzweigt werden soll. Auf die gleiche Art und Weise wird die absolute Speicheradresse bei Programmierung des QL im Assembler durch symbolische Adressen ersetzt (s. a. Kapitel 8). Der Vorteil der symbolischen Adresse ist klar ersichtlich: Falls die Zeilennummern im Programm Abb. 2.2 geändert werden, müssen auch die Adressen der GOTO-Befehle überprüft und eventuell angepaßt werden. Bei dem Programm in Abb. 2.6 ist das nicht nötig, da die Zeilennummern keine Referenzpunkte sind.

Die Begrenzung von Schleifen und Strukturen

Nicht nur die FOR- und REPeat-Schleifen sind eindeutig und übersichtlich abgegrenzt, sondern, wie wir gesehen haben, auch die IF-Konstruktion. SuperBASIC ist das beste Beispiel für eine interaktive Sprache auf einem Mikrocomputer, die diese Methode der Begrenzung von Strukturen benutzt. Durch den Einschluß derartiger Konstruktionen in „Klammern" wird der Umfang dieser Programmteile absolut eindeutig. Die ursprüngliche und immer noch die beste Version dieser strukturierten Programmierung war die Spezifikation der überlegenen Programmsprache ALGOL 68.

Die bis jetzt bekannten SuperBASIC-„Strukturklammern" sind:

```
REPeat      END REPeat
FOR         END FOR
IF          END IF
```

Die Art, wie in SuperBASIC mit IF... THEN... ELSE... END IF gearbeitet wird, ist der PASCAL-Version des IF... THEN... ELSE sehr ähnlich. PASCAL ist eine bekannte Programmiersprache akademischen Ursprungs und wird als Beispiel für leicht verständliche, strukturierte Programmierung hoch geschätzt.

Untersuchen wir die PASCAL-Anweisung:

```
IF bedingung  THEN aktion 1  ELSE aktion 2;
```

Sie führt dazu, daß „aktion 1" ausgeführt wird, wenn die „bedingung" erfüllt (true) ist, anderfalls wird „aktion 2" ausgeführt.

Das Gegenstück in ALGOL 68 ist:

```
IF bedingung  THEN aktion 1  ELSE aktion 2  FI;
```

wobei FI (die Umkehrung von IF) das Ende der IF-Anweisung markiert. Das Semikolon wird in ALGOL 68 lediglich als Zeichen für das Ende eines Statements verwendet, nicht aber für die Begrenzung von Konstruktionen.

In SuperBASIC würde die Anweisung – ohne die Zeilennummern zu beachten, die ohnehin lediglich der Programmpflege dienen – wie folgt geschrieben:

```
IF bedingung         THEN
                     aktion 1
ELSE
                     aktion 2
END IF
```

Ein Format, das sehr übersichtlich und leicht zu lesen ist. Die ALGOL 68- und SuperBASIC-Versionen (IF......FI und IF......END IF) sind dem PASCAL-Format überlegen. Es ist bezeichnend, daß der neue Standard von FORTRAN 77 ebenfalls ein END IF verwendet.
Als Beweis dafür, daß die IF-Handhabung nicht so eindeutig ist, sollten Sie die folgende PASCAL-Anweisung lesen:

```
IF bedingung 1  THEN IF bedingung 2  THEN aktion 1
ELSE aktion 2;
```

Es ist nicht sofort ersichtlich, zu welchem IF das ELSE gehört. Der PASCAL-Übersetzer führt diese Anweisung in einer speziellen Art aus, die in ALGOL 68 so kodiert wird:

```
IF bedingung 1   THEN
             IF bedingung 2  THEN aktion 1  ELSE aktion 2
FI
```

Im Gegensatz zur ALGOL-Anweisung ist die PASCAL-Anweisung nicht so klar und übersichtlich; man muß bei PASCAL wissen, wie der Übersetzer arbeitet. Diese Unklarheit ist in PASCAL (und in einigen ähnlichen Sprachen) wohl bekannt, sie wird vielfach als das „Problem des hängenden ELSE" bezeichnet. In SuperBASIC gibt es kein „hängendes ELSE", denn die Bedeutung von

```
IF bedingung 1  THEN
             IF bedingung 2  THEN
                             aktion 1
             ELSE            aktion 2
             END IF
END IF
```

ist ebenso klar ersichtlich wie die der folgenden Konstruktion:

```
IF bedingung 1   THEN
                 IF  bedingung 2  THEN aktion 1
ELSE             aktion 2
END IF.
```

So könnte auch die o.a. PASCAL-Anweisung verstanden werden. Das ist aber falsch. Diese Konstruktion muß in PASCAL im folgenden Format geschrieben werden:

```
IF bedingung 1    THEN
                 BEGIN  IF  bedingung 2  THEN aktion 1  END
                 ELSE  aktion 2
```

Aber auch dieses Format ist nicht sehr übersichtlich. Die zweite Version der Anweisung könnte für den QL in SuperBASIC (über mehr Zeilen) auch wie folgt geschrieben werden:

```
IF bedingung 1    THEN
                 IF bedingung 2  THEN
                                 aktion 1
                 END IF
ELSE             aktion 2
END IF
```

Es kann unterstellt werden, daß das BEGIN ... END Paar eine Art von Klammern für PASCAL-Konstruktionen ist. Leider beutzt die Programmiersprache PASCAL diese Klammern nicht konsequent für alle derartigen Konstruktionen, deshalb ist PASCAL nicht so logisch zusammenhängend und ähnlichen Sprachen insofern unterlegen.

Man kann die Struktur und Logik einer IF... THEN... ELSE... END IF Konstruktion durch Verwendung von zwei Kontrollmechanismen analysieren, IMMER und FALSCH. Nehmen wir eine Konstruktion des Formats

```
IF bedingung  THEN
              aktion 1
ELSE          aktion 2
END IF
```

Wenn hier die Bedingung nicht erfüllt ist (FALSCH), geht die Kontrolle auf das ELSE-Segment über, andernfalls wird die „aktion 1“ aktiviert. Wenn diese vollständig ausgeführt ist, geht die Programmkontrolle IMMER auf die Anweisung nach END IF über.
Die Logik dieser Konstruktion ist in Abbildung 2.7 schematisch dargestellt.

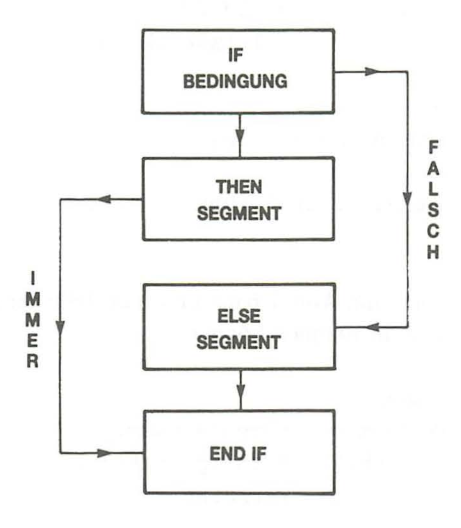


Abb. 2.7 Das IF Ablauf-Diagramm

SuperBASIC verwendet die END-Klammer nicht nur bei REPeat, FOR und IF, sondern noch bei einer weiteren Konstruktion: bei der SELect-Anweisung, die aus einer Reihe von bedingten Verzweigungen oder Anweisungen besteht. Die SELECT-Konstruktion ist den anderen bedingten Konstruktionen sehr ähnlich (zum Beispiel benutzen alle FALSCH und/oder IMMER).

Das Format der SELECT-Konstruktion (eine ausgefeiltere Version der CASE-Konstruktion in PASCAL) ist:

```
SELect    ON schalter
             ON schalter-bedingung-1
                         aktion 1
             ON schalter-bedingung-2
                         aktion 2
             .........   (weitere ON-Anweisungen)
             ON schalter=REMAINDER
                         letzte aktion
END SELect.
```

Ebenfalls gültig ist das einzeilige Format:

```
SELect  ON  schalter-bedingung : aktion
```

Im SuperBASIC-Programm Abb. 2.6 hätten wir anstelle der IF-Konstruktion auch die SELect-Struktur nehmen können:

```
SELECT  ON nummer
            ON nummer = computernummer
                 PRINT „Richtig nach"; versuche;
                       „Versuchen"
                 STOP
            ON nummer = computernummer TO 100
                 PRINT „Ihre Zahl ist zu gross"
            ON nummer = 0 TO computernummer
                 PRINT „Ihre Zahl ist zu klein"
            ON nummer = REMAINDER
                 PRINT „Falsche Eingabe"
END SELect
```

Die Prüfungen in diesem Programmabschnitt sind etwas raffinierter als im vorigen Programm. Es wird nicht nur nach Gleichheit gefragt (ON nummer = computernummer), sondern es werden auch die Limits der Eingabe überprüft (nummer = REMAINDER).

Beachten Sie, daß keine Zeilennummern vergeben werden, da sie für Super-BASIC bedeutungslos sind. Alle folgenden Programmbeispiele werden deshalb ebenfalls ohne Zeilennummern geschrieben. Sie können, bei jeder beliebigen Zeilennummer beginnend, vergeben werden und werden von Super-BASIC problemlos verkraftet, solange sie in aufsteigender Folge sind. An dieser Stelle soll auf einen interessanten Punkt hingewiesen werden: PASCAL-Programme werden im allgemeinen ohne Zeilennummer dargestellt, untersucht man jedoch wirkliche Programmlisten, dann findet man an jeder Zeile eine Nummer, allerdings nur für die Referenz bei der Programmpflege.

Die Kontroll-Struktur der SELect-Konstruktion ist recht einfach zu bestimmen, deshalb überlasse ich die Erstellung des Diagramms dem Leser als kleine Übungsaufgabe.

IMMER und Rekursion

Ein anderer Weg der Betrachtung des Ratespiels HILO wäre folgender: Raten heißt eine Zahl auswählen. Ist die Zahl falsch, dann wird wieder geraten. Natürlich muß dieser Prozeß zum Ende kommen, wenn die gewählte Zahl richtig ist.

Die in diese Betrachtungsweise implizierte Struktur entspricht genau dem Diagramm in Abb. 2.3, und die folgende Programmierung dieses Schemas erfolgt mit Hilfe einer anderen Konstruktion, die ebenfalls ein END hat. Man nennt diese Struktur eine Prozedur, und der Bereich der Prozedur wird abgegrenzt durch DEFine...... und END DEFine.

```
DEFine PROCedure prozedurname
              (evtl. zusätzliche Parameter)
       ausführbare Anweisungen 1
       bedingter RETURN
       ausführbare Anweisungen 2
END DEFine.
```

Damit kann unser Ratespiel in die elegante kleine Prozedur „spiel“ umgeschrieben werden.

```
DEFine PROCedure spiel (computernummer, versuche)
   LOCAL nummer
   INPUT „Welche Zahl raten Sie"; nummer
   SELECT ON nummer
        ON nummer = computernummer
               PRINT „Richtig nach"; versuche;
               „Versuchen"
               RETURN
        ON nummer = computernummer TO 100
               PRINT „Ihre Zahl ist zu gross"
        ON nummer = 0 TO computernummer
               PRINT „Ihre Zahl ist zu klein"
        ON nummer = REMAINDER
               PRINT „Falsche Eingabe"
END SELect
spiel (computernummer, versuche + 1)
```

Abb. 2.8 Das HILO-Programm in SuperBASIC als Prozedur geschrieben

Diese Prozedur kann in einem Programm verwendet werden. Es ist aber auch möglich, sie interaktiv mit dem Aufruf (= der Eingabe)

```
spiel (zufall, 1)
```

zu aktivieren. Aufgrund dieser Eingabe wird zunächst die Prozedur „spiel" als Programm aufgerufen, sodann der Randomwert „zufall" als Konstante „computernummer" übernommen und die Variable „versuche" auf 1 gesetzt.

Der Prozedur-Ablauf kann durch die Definition einer zusätzlichen Prozedur HILO (o. ä.) noch vereinfacht werden:

```
DEFine PROCedure HILO
        RANDOMISE
        spiel (zufall,1)
END DEFine
```

Jetzt wird durch die Eingabe HILO die RANDOMISE Prozedur und anschließend die Prozedur „spiel" aufgerufen. Die Prozedur „spiel" benutzt die Funktion „zufall" als ersten Parameter und den Wert 1 als zweiten.

Die Prozedur „spiel“ hat intern zwei weitere Parameter: „computernummer“ und „versuche“. Außerdem wird die Variable „nummer“ LOCAL benutzt, das heißt, daß andere Variablen mit dem Namen „nummer“ außerhalb dieser Prozedur benutzt werden können, ohne daß die gleichen Bezeichner zu Fehlern führen. Alle anderen Bezeichner „nummer“, die außerhalb dieser Prozedur liegen, werden von der Prozedur „spiel“ ignoriert.

Wir haben INPUT und SELect schon vorher verwendet, es muß aber betont werden, daß im Gegensatz zum STOP am Ende eines normalen Programms eine Prozedur mit einem (impliziten) RETURN endet. Dieser RETURN ergibt sich aus dem END DEFine der Prozedur.

Die Benutzung des SELect entspricht den bisherigen Anwendungen, aber nach dem END SELect finden wir eine interessante Zeile:

```
spiel (computernummer, versuche+1)
```

Die von uns untersuchte Prozedur heißt „spiel“, und innerhalb dieser Prozedur „spiel“ wird eine Prozedur aufgerufen:

```
spiel (computernummer, versuche+1).
```

Die Prozedur ruft sich also selber auf. Ein derartiger Aufruf ist ein rekursiver Aufruf.
Wenn eine Prozedur sich selber aufruft, dann arbeitet sie mit den neuen Parameterwerten; mit den Werten, die sich aus dem letzten Durchlauf ergeben haben. In unserem Beispiel heißt das, daß der Inhalt von „versuche“ um den Wert 1 erhöht wird. Das läßt sich am einfachsten mit der Prozedur „erhöhe-um-eins“ verdeutlichen:

```
DEFine  PROCedure  erhöhe-um-eins(variable)
        PRINT variable
        erhöhe-um-eins(variable+1)
END DEFine
```

Durch die Eingabe von ‚erhöhe-um-eins(5)‘ oder
‚erhöhe-um-eins 5‘
wird diese Prozedur aufgerufen und SuperBASIC antwortet mit der Ausgabe von

```
5
6
7
8
9
10
```

und so weiter, bis die Tasten CTRL, ALT und LEERSCHRITT gleichzeitig gedrückt werden.

Die Prozedur erhöhe-um-eins hat nur einen Parameter, der in der Definition „variable“ genannt wurde. Beim Aufruf der Prozedur kann der Anfangswert sowohl in Klammern: erhöhe-um-eins(5) oder nur durch einen Leerschritt getrennt: erhöhe-um-eins 5 vorgegeben werden.
Beim Aufruf der Prozedur wird der formale Parameter (genannt variable) durch den eingegebenen Wert (im Beispiel: 5) ersetzt. Bei jedem Durchlauf durch diese Prozedur wird der formale Parameter durch den aktuellen Wert ersetzt. Beim ersten Aufruf von „erhöhe-um-eins“ führt deshalb der Befehl: ‚PRINT variable‘ zur Ausgabe der Ziffer 5, dem eingegebenen Anfangswert für „variable“.

In der Prozedur „erhöhe-um-eins“ haben wir den Aufruf erhöhe-um-eins (variable+1).

Für diesen Aufruf ist der aktuelle Wert = 6 (5+1) und so wird die Prozedur „erhöhe-um-eins“ erneut durchgeführt. Für SuperBASIC sind die beiden Aufrufe von „erhöhe-um-eins“ zwei getrennte Vorgänge, und der Super-BASIC-Übersetzer muß über beide die Kontrolle behalten.

Bei der neuen Aktivierung von „erhöhe-um-eins“ ist der aktuelle Wert von „variable“ = 6, deshalb führt der Befehl: PRINT variable zur Ausgabe der Ziffer 6.

Wenn wir die zweite Aktivierung des Parameters „variable“ als „variable:2“ bezeichnen (im Gegensatz zu „variable:1“ vom ersten Durchlauf), dann ergibt sich „variable:3“ aus „variable:2+1“.

Achtung: in SuperBASIC ist die Verwendung von Bezeichnern wie variable:1 oder variable:2 nicht zulässig!

Die fortgesetzte Erhöhung des Wertes für den Parameter „variable“ produziert eine ständig erhöhte Ausgabe: wir erhöhen um eins. Diese Steigerung

des Wertes wird nur beendet, wenn die Zahl für den QL zu groß wird (was angesichts des Umfangs der Zahlen, die SuperBASIC auf dem QL erlaubt, unwahrscheinlich ist), oder der Speicher des QL ist voll ausgeschöpft. Denn bei jedem Aufruf von „erhöhe-um-eins“ wird für die Kontrolle dieses Aufrufs etwas Speicherraum beansprucht, diese Werte werden in Erwartung des END DEFine gespeichert.
Natürlich ist es schneller (und sinnvoller), das Programm durch Anschlag der entsprechenden Kontrolltasten zu beenden, also CTRL, ALT und Leertaste.

Definition von Funktionen

Unterstellen wir, daß wir eine mathematische Funktion (genannt SGN) definieren wollen, die nach Bearbeitung einer Zahl die folgenden Ergebnisse bringt:

```
IF x < 0 THEN SGN(x)= -1
IF x = 0 THEN SGN(x)= Ø
IF x > 0 THEN SGN(x)= +1
```

Das mag für Sir Clive Sinclair eine Kleinigkeit sein, für den QL-Benutzer ist es ein Riesenschritt vorwärts.

Hier ist der erste Versuch:

```
DEFine FuNction SGN(x):REM  Version 1
       LOCAL   ergebnis
       LET  ergebnis = x/ABS(x)
       RETURN  ergebnis
END DEFine
```

Die Idee hinter dieser Funktion ist: Die Division eines negativen Wertes in der Variablen X durch seinen absoluten Wert ergibt −1. Ist X positiv, dann ist auch das Vorzeichen positiv. Die DEFinition dieser Funktion, so kurz wie sie ist, hat eine Reihe bezeichnender Aspekte (außer der Tatsache, daß sie nicht arbeitet, wenn X = 0 ist).

Der formale Parameter ist X, außerdem wurde eine LOCAL Variable „ergebnis“ definiert. Diese LOCAL Variable ist nur innerhalb dieser Funktion ansprechbar, andere Variablen außerhalb dieser Funktion mit dem gleichen Namen werden durch die Benutzung der LOCALen Variablen „ergebnis“ nicht verändert.

Der großzügige Gebrauch von LOCALen Bezeichnern erspart Komplikationen, die sich aus mehrfacher Verwendung gleicher Bezeichner ergeben (allzuschnell hat man mehrere Variablen „ergebnis“).

Der Variablen „ergebnis“ wird der Wert X/ABS(X) zugewiesen und der Wert der Variablen „ergebnis“ wird als Ergebnis der Funktion RETURNed (= zurückgegeben). (In einer einzeiligen Funktions-Definition folgt der Wert dem „=“-Zeichen.)

Wie bereits bemerkt wurde, führt X=O zur Division „ergebnis“=0/0 und damit zu einem Überlauf-Fehler. Deshalb gehört in die Funktion eine Prüfung auf X=0.

```
DEFine FuNction SGN(X):REM  Version 2
       IF X = 0 THEN RETURN 0
       RETURN X/ABS(X)
END DEFine
```

Diese zweite Version benutzt nicht die Möglichkeit der LOCALen Variablen. Sie ist zudem nicht nur kompakter, sie ist auch korrekt.

Es ist zu beachten, daß der Umfang einer Funktion ebenso klar abgegrenzt ist wie bei Schleifen, trotzdem besteht die Möglichkeit eines vorzeitigen Ausgangs. Bei den Schleifen benützen wir die Anweisung EXIT, wogegen bei Prozeduren und Funktionen ein RETURN erfolgt (obwohl die genaue Untersuchung des RETURN bei Prozeduren und Funktionen gewisse Unterschiede zeigt).

Der Bubble Sort („Blasen"-Sortierung)

Unterstellen wir, wir hätten eine Reihe von unsortierten Zahlen, so zum Beispiel:

```
7
1
5
3
2
4
6
```

und wir wollen diese Zahlen in aufsteigender Folge ausgeben, also mit 1 beginnend und die höchste Zahl zuletzt. Der Bubble Sort ist eine Möglichkeit, diese Zahlen zu sortieren; zwar nicht die effizienteste, aber immerhin die am leichtesten verständliche.

Der Bubble Sort beginnt mit dem Austausch jeweils nacheinander stehender Elemente, wenn diese in verkehrter Ordnung stehen:

```
7 1 1 1 1 1 1
1 7 5 5 5 5 5
5 5 7 3 3 3 3
3 3 3 7 2 2 2
2 2 2 2 7 4 4
4 4 4 4 4 7 6
6 6 6 6 6 6 7
```

So verläuft der erste Durchgang durch die Werte, wobei der höchste Wert wie eine Blase durch die niedrigeren Werte nach unten gleitet. Hier ein weiterer Durchlauf:

```
1 1 1
3 3 3
5 2 2
2 5 4
4 4 5
6 6 6
7 7 7
```

Obwohl die Suche weiterläuft, werden keine weiteren Änderungen in diesem Durchlauf vorgenommen.

Der nächste Durchlauf ergibt sich nach dem Austausch von 3 und 2

```
1
2
3
4
5
6
7
```

und die Sortierung ist beendet.

Lassen Sie uns untersuchen, was zu geschehen hat.

A Vergleiche die Werte in der Liste und tausche die in der verkehrten Reihenfolge stehenden Werte mit dem Wert davor.
B Wenn Elemente ausgetauscht wurden, dann wiederhole Aktion A.

Jetzt können wir eine Funktion definieren, wo „liste" eine Tabelle mit einer bestimmten Anzahl von Elementen ist, nämlich

Anzahl der Elemente minus Eins.

```
DEFine PROCedure bubble-sort
       LOCAL index, flag-tausch
       LET flag-tausch=0
       FOR index = 0 TO elementeanzahl -1
           IF liste(index)> liste(index +1) THEN
              tausche-elemente(index)
              LET flag-tausch = 1
           END IF
       END FOR
       IF flag-tausch = 1  THEN bubble-sort
END DEFine
```

Jetzt müssen wir noch die Prozedur „tausche-elemente" definieren. Das könnte so aussehen:

```
DEFine PROCedure  tausche-elemente(zähler)
       LOCAL zwischenspeicher
       LET zwischenspeicher = liste(zähler)
       LET liste(zähler) = liste(zähler +1)
       LET liste(zähler +1)=zwischenspeicher
END DEFine
```

Hierbei wird die Tabelle „liste“ von beiden Prozeduren angesprochen (sie ist also auch außerhalb der Prozeduren veränderbar!).

Die Prozedur „tausche-elemente“ ist leicht zu verstehen (und ist in mehreren Versionen im Maschinencode programmiert worden, s.a. Kapitel 8); interessanter ist die Prozedur „bubble-sort“.

Bezeichner und Coercion

Die Prozedur „bubble-sort“ könnte wie folgt in einem Programm benutzt werden:

```
LET  elementeanzahl = 6
DIMension liste(elementeanzahl)
LET liste = {7,1,5,3,2,4,6}
bubble-sort
FOR element = 0 TO 6 : PRINT liste(element)
STOP
```

Dieses dargestellte Programm enthält einige interessante Punkte. Einer davon ist die Anweisung:

LET liste = {7,1,5,3,2,4,6},

die die Wichtigkeit und Bedeutung der Idee der „bezeichner“ in SuperBASIC verdeutlicht. Wenn der SuperBASIC-Übersetzer eine Reihe von Zeichen findet (zum Beispiel „liste“), dann analysiert und erkennt er, was der Bezeichner beinhaltet.

Der SuperBASIC-Übersetzer erkennt „liste“ als eine Tabelle und die Anweisung:

LET liste = {7,1,5,3,2,4,6}

wird als Anweisung zum Anlegen der Tabelle „liste“ verstanden und ausgeführt. Allerdings muß diese Tabelle für das System vorher deklariert werden, das geschieht mit der Anweisung DIMension.

In BBC BASIC, zum Beispiel, müßte eine Prozedur mit dem Namen „bubblesort“ in folgender Form angesprochen werden:

PROC-bubble-sort.

Da SuperBASIC die Schlüssel-Idee der „bezeichner“ hat, würde dieses Format der Logik der „bezeichner“ widersprechen. „bezeichner“ dienen der Identifikation und SuperBASIC kümmert sich darum, was dahintersteckt. So entsprechen die formalen Parameter z. B. für Funktionen und Prozeduren keinem bestimmten Typ. Sie sind in ihrer Natur nicht unbedingt vorgegeben. SuperBASIC nimmt die aktuellen Parameter und versucht auf jeden Fall etwas „Vernünftiges“ aus den Parametern zu einem „bezeichner“ zu machen.

Dieses Bemühen, aus den „bezeichnern“ in einer Anweisung etwas Vernünftiges zu produzieren, läßt sich am besten an folgendem kleinen Programm demonstrieren:

```
LET A$ = "1"
LET B$ = "2"
LET C$ = "3"
LET D$ = A$ + B$ + C$
LET D$ = "A" & D$
PRINT D$
```

Die Ausgabe durch den PRINT-Befehl ist A6.

Dieses Programm veranschaulicht sehr deutlich das ebenfalls sehr wichtige Konzept der Coercion. (Die Möglichkeiten von Coercion und Bezeichnern sind scheinbar erst durch die Programmiersprache ALGOL 68 in das Blickfeld der Computerwissenschaft gerückt worden).
Coercion bedeutet, daß die Werte der Bezeichner entsprechend den Operationen und dem Kontext (z.B. + wird für die Addition von Zahlen verwendet) soweit als möglich in die für die Operationen akzeptable Form gezwungen (= coerced) werden.

Die Alphafelder A$, B$ und C$ enthalten Zeichen, die in Zahlen umgeformt werden können (bei einer Anweisung wie LET A$ = “P” ist diese Umwand-

lung – Coercion – kaum zu rechtfertigen und durchzuführen). Die Operation + erwartet Zahlen, deshalb werden die Zeichen in die entsprechenden Zahlen umgewandelt und anschließend addiert; das Ergebnis ist der Wert 6. Der Bezeichner auf der linken Seite (also D$) ist ein Alphafeld, deshalb wird der Wert in das Zeichen "6" umgewandelt. Anschließend wird das Zeichen "A" mit dem Feld D$ verbunden und in der Ausgabe erscheint „A6".

Die Möglichkeiten der Coercion sind weitreichend; möglicherweise umfangreicher, als es die Designer von SuperBASIC ursprünglich beabsichtigten.

Die Verfasser von SuperBASIC haben versucht, die Sprache so logisch und voraussehbar zu machen wie es nur irgend möglich war. Deshalb ist im allgemeinen durchaus abzusehen, was geschieht, wenn bestimmte Effekte programmiert werden. Aber mit solchen kräftigen Werkzeugen wie Coercion und Bezeichnern verfügt SuperBASIC über ein Potential, das noch lange nicht ausgeschöpft ist.

Noch einmal Blasen

Lassen Sie uns nach dieser Abschweifung über Bezeichner und Coercion zu unserem Problem zurückkehren: wir wollten ein effizientes Bubble Sortier-Programm schreiben.

Bei der schrittweisen Betrachtung einer Bubble Sortierung haben wir festgestellt, daß der größte Wert immer wie eine Blase durch die niedrigeren Werte schwimmt, so daß wir nach dem ersten Durchlauf nicht mehr sieben, sondern nur noch sechs Elemente zu berücksichtigen hatten. Nach dem zweiten Durchlauf waren es nur noch fünf, und so weiter. Das Sortierprogramm benötigt also einen Parameter für die Anzahl der jeweils zu berücksichtigenden Elemente.

```
DEFine PROCedure  bubble-sort(elementeanzahl)
   LOCAL index, flag-tausch
   LET  flag-tausch = Ø
   FOR index = 0 TO elementeanzahl -1
               IF liste (index) > liste (index +1)  THEN
                    tausche-elemente(index)
                    LET flag-tausch = 1
               END IF
```

```
    END FOR index
    IF  flag-tausch = 1  THEN
        bubble-sort (elementeanzahl -1)
    END IF
END DEFine
```

Damit beschleunigen wir die Sortierung etwas. Eine weitere Beschleunigung kann durch die Berücksichtigung des Tausches erzielt werden. Wenn die Elementnummer, bei der der Tausch vorgenommen wird, kleiner als der Wert in anzahl-elemente ist, dann kann der Durchlauf verkürzt werden.

Im obigen Beispiel wurden im ersten Durchlauf die Elemente an den Positionen sechs und sieben getauscht. Die Suche im nächsten Durchlauf braucht also nur bis Position sechs zu gehen (obwohl das in unserem Beispiel nicht einmal nötig ist). Im zweiten Durchlauf werden die Elemente vier und fünf getauscht, kein größeres. Aus diesem Grund braucht der nächste Durchlauf auch nur bis Element vier zu gehen. Hier ist die verbesserte Version:

```
DEFine  PROCedure  bubble-sort (anzahl-elemente)
   LOCAL index, flag-tausch
   LET flag-tausch = 0
   FOR index = 0 TO anzahl-elemente -1
       IF  liste (index) > liste (index +1)  THEN
                     tausche-elemente (index)
                     LET flag-tausch = index
       END IF
   END FOR index
   IF flag-tausch > 0  THEN  bubble-sort (flag-tausch)
END DEFine
```

Auch wenn der Bubble-Sort eine triviale Angelegenheit ist, er zeigt sehr deutlich die Stärke und Flexibilität von SuperBASIC.

3. SuperBASIC Grafik

Bevor wir die Grafik von SuperBASIC diskutieren, sollten wir bestimmte Begriffe der Grafik und der Erzeugung von grafischen Effekten klar definieren. Als erstes müssen wir feststellen, was Punkte, Linien und Flächen sind.

Punkte und Pixels

In der Geometrie ist ein Punkt lediglich eine Stelle im Raum, etwas ohne Größe oder Form, lediglich eine Position. Euklid definierte den Punkt als „etwas, das keine Teile hat". Ein Punkt hat eine Position, aber keine Größe (Chambers, Lexikon des zwanzigsten Jahrhunderts). Für Euklid und andere alte Griechen (aber auch für moderne Mathematiker) repräsentiert der Begriff Punkt ein theoretisches Konzept, abstrahiert aus der Realität.

In der Schule haben wir in Geometrie gelernt, daß der Punkt etwas Mysteriöses, Geheimnisvolles ist. Es gibt Beweise dafür, daß auf jeder Linie eine unendliche Anzahl von Punkten liegt (z.B. Eugene P. Northrop, Rätsel in der Mathematik), und auf jedem Teil einer Linie liegen soviele Punkte wie auf der ganzen Linie. Die Mathematiker haben einen geradezu perversen Stolz, die Schwächen des Punktes zu beweisen.

Betrachten Sie einen beliebigen Punkt, den Sie gezeichnet haben, und Sie werden zweifelsfrei feststellen, daß Ihr Punkt eine bestimmte Position hat (genau genommen sogar mehrere), und ganz bestimmt hat Ihr Punkt auch eine Größe. Hätte er keine Größe, dann wäre er unsichtbar. Selbst das unsichtbare Atom hat eine Größe (auch wenn sie keiner kennt). Wenn man die abstrakte Welt der Mathematik verläßt, dann belegen Punkte immer etwas Platz oder Fläche.

Geben Sie dieses kurze Programm für den QL ein und lassen Sie es ausführen.

Sie werden dann einen Punkt auf dem Bildschirm sehen, er erscheint da als ein Fleckchen.

```
AT RND(2 TO 10), RND(2 TO 20):PRINT"."
```

Untersuchen Sie den Punkt (auf dem Bildschirm) genau. Wenn Sie einen Monitor benutzen, werden Sie erkennen, daß der Punkt aus mehreren Monitor-Punkten besteht, sollten Sie einen Fernseher als Monitor benutzen, dann ist der Punkt wahrscheinlich etwas unscharf.

Der QL hat seine eigenen Punkte, sie werden Pixel genannt. Er hat zwei verschiedene Grafik/Punkt-Auflösungen, also zwei verschiedene Punkt-Größen auf dem Bildschirm. Die kleinste Auflösung (die kleinsten Punkte), die der QL schafft, ist 512 mal 256 Pixel und wird auch 512-Pixel-Mode genannt. Eine gröbere Auflösung (also größere Punkte) bringt der 256-Pixel-Mode (die Auflösung beträgt hier 256 mal 256 Pixel).
Die Anzahl Pixel auf einer Linie quer über den Bildschirm kann also 512 oder 256 betragen, denn bei der Angabe der Pixelzahlen steht immer die Anzahl Pixel quer über dem Bildschirm (also in der X-Richtung) an erster Stelle, gefolgt von der Anzahl Pixel in der Senkrechten. Die niedrigste Auflösung hat der normale Text, obwohl hier die Auflösung in Abhängigkeit von der Buchstabengröße variieren kann.

Wenn man auf zwei gleichgroßen Bildschirmen die beiden Auflösungen vergleicht, kann man feststellen, daß bei der feinsten Auflösung die Pixel halb so groß sind wie bei der mittleren Auflösung. Ein Pixel ist in der mittleren Auflösung zweimal so dick wie ein Pixel der feinsten Auflösung. Hierdurch wird das Verständnis für Zeichen (der gröbsten Auflösung) etwas erleichtert.

Die Zeichen auf dem Bildschirm können verschiedene Größen haben, die Größe wird durch die Pixelgröße bestimmt. Jedes Zeichen kann eine von vier verschiedenen Breiten und eine von zwei verschiedenen Höhen haben. Die Zeichengröße wird durch

```
CSIZE [freie-kanal-nummer,] Breiten-Code, Höhen-Code
```

definiert. Die Codes sind in den Tabellen 3.1 und 3.2 dargestellt.

Tabelle 3.1 Breiten-Codes für die Zeichen

Code	Größe	256-Pixel-Mode	512-Pixel-Mode
0	6 Pixel	42 Spalten	85 Spalten
1	8 Pixel	32 Spalten	64 Spalten
2	12 Pixel	21 Spalten	42 Spalten
3	16 Pixel	16 Spalten	32 Spalten

Tabelle 3.2 Höhen-Codes

Code	Größe	Zeilen
0	10 Pixel	25
1	20 Pixel	12

Die Default-Werte für den 256-Pixel-Mode sind 0,0 und ergeben 25 Zeilen und 40 Zeichen (das erlaubt etwas Flexibilität), und die Default-Werte für den 512-Pixel-Mode sind 2,0, was ebenfalls 25 Zeilen je ca. 40 Zeichen ergibt. Wenn das Zeichen-Format im 512-Pixel-Mode durch die Anweisung:

```
CSIZE 0,0
```

geändert wird, dann werden 80 Zeichen pro Zeile dargestellt. Auf einem normalen Fernseher (UHF Empfänger) ist es aber sehr schwierig, eine 80-Zeichen-Zeile zu lesen.
Für die QL-Benutzer ohne Video-Monitor, nur mit einem normalen Fernseher als Monitor, ist der 512-Pixel-Mode sinnlos (und bestimmte Effekte, wie z.B. Punktmalerei, sollten mit UHF-Empfängern gar nicht erst versucht werden).

Zur Spezifikation von bestimmten Pixeln wird das „Pixel-Koordinaten-System“ benutzt (siehe Abb. 3.2), und in diesem System wird immer der 512-Pixel-Modus erwartet. Wenn das System im 256-Pixel-Modus arbeitet, gilt das gleiche Koordinaten-System, jedoch wird der dem 256-Pixel-Mode am nächsten verfügbare Pixel benutzt.
Das Pixel-Koordinaten-System ist anders als das Grafik-Koordinaten-System. Beide Systeme wurden entwickelt, um Sie dort vor Platzverschwendung zu schützen, wo all das gespeichert ist, was auf dem Bildschirm passiert; und da passiert viel.

Der Platz, wo diese ganzen Informationen gespeichert sind, ist natürlich der Speicher des QL. Der QL benutzt eine Grafik-Version, die „Bit-gespeicherte“ Grafik oder „Raster“ Grafik genannt wird.
Sie wird „Bit-gespeicherte“ Grafik genannt, weil jedes Pixel auf dem Bildschirm der Anordnung der Bits im Speicher entspricht. Der Begriff „Raster-Grafik“ kommt aus der Standard TV Bildschirmtechnik, wo das Bild durch Punkte verschiedener Intensität aufgebaut wird. Die Zeilen auf dem Bildschirm werden vom TV-Raster zerlegt, von rechts nach links und von oben nach unten (625 Zeilen bei UHF-Empfängern), um das Bild aufzubauen.

Gleichgültig, wie die Art der Darstellung genannt wird, es bedeutet, daß die grafische Darstellung auf dem Bildschirm (bei ausreichender Übung) durch die Manipulation des Inhalts bestimmter Speicher-Adressen ohne spezielle Befehle kontrolliert wird. Trotzdem existiert für den QL (wie auch beim BBC Computer) ein Satz grafischer Prozeduren in SuperBASIC, der die Arbeit mit Grafik so erleichtert, daß die Benutzung des Maschinen-Codes völlig unnötig ist.

Zeilen und Kurven

Versuchen sie dieses kleine Programm:

```
FOR spalte = 2 TO 38
    LET zeile = INT(55*spalte+0.5)
    AT spalte, zeile:PRINT "*"
END FOR
```

das Sterne waagerecht und senkrecht auf dem Bildschirm zeichnet. Ziehen Sie auf Millimeter-Papier die Linie

```
X = 0,55·Y,
```

und Sie werden sehen, daß diese Linie ganz gerade und glatt ist (ist sie es nicht, dann ist sie schlecht gezeichnet).
Die unscharfe Linie aus Sternen ist (in niedriger Auflösung) das Gegenstück der glatten Linie der mathematischen Grafik. Euklid sagte: eine Linie ist das, „was gleichmäßig zwischen den Endpunkten liegt“. Aber nur wenige Linien auf Computer-Bildschirmen liegen glatt zwischen den Enden. So wie ein

Punkt auf dem Computer eine bestimmte Größe haben muß (das Pixel), so besteht eine Linie auf dem Bildschirm aus möglichst geraden Linien mit rechten Winkeln dazwischen.

Wie man bei der Linie in geringer Auflösung sieht (die übrigens aus etwas größerer Entfernung besser aussieht), sind Computerlinien nicht ganz gerade, sondern werden aus kurzen Linien von waagerechten und senkrechten Pixeln gebildet. Abbildung 3.1 zeigt das deutlich. Wenn Sie die Augen etwas zukneifen und nicht zu nahe sind, wirkt die Linie gerader.

```
******
          ******
                    ******
                              ******
                                        ******
                                                  ******
                                                            ******
```

Abb. 3.1 Eine (fast) gerade Linie

Linien werden in SuperBASIC üblicherweise mit dem LINE-Befehl gezogen. Es gibt davon zwei Haupt-Varianten:

```
LINE X1,Y1 TO X2,Y2
LINE TO X3,Y3
```

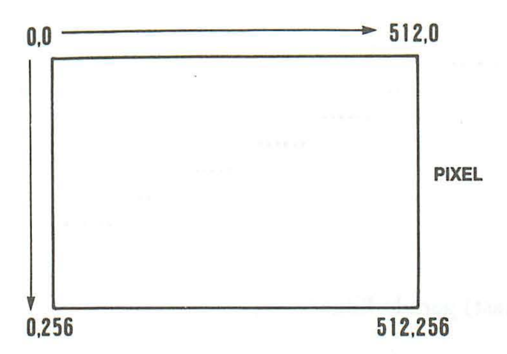
Bei der ersten Variante wird unter Verwendung der grafischen Koordinaten (nicht: Pixel-Koordinaten) eine Linie von den Koordinaten X1,Y1 zu den Koordinaten X2,Y2 gezogen. Beim zweiten Befehl wird vom augenblicklichen Stand des grafischen Cursors (wie immer der aussieht) eine Linie zu den Koordinaten X3,Y3 gezogen.

Eine weitere Variante ist

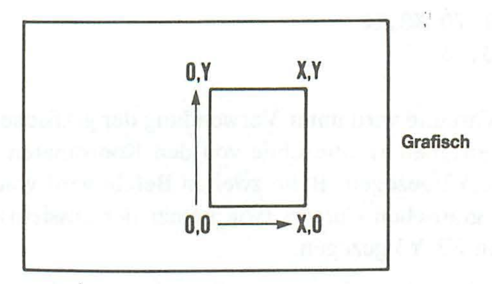
```
LINE X1,Y1 TO X2,Y2 TO X3,Y3
```

die um weitere Koordinaten erweitert werden kann.

Abbildung 3.2 stellt Pixel- und Grafik-Koordinaten-Systeme gegenüber und demonstriert die Unterschiede. Nicht nur, daß die beiden Systeme in der Handhabung unterschiedlich sind, sie sind auch vom Konzept her inkompatibel.
Das Pixel-Koordinaten-System ist ein festes System von Koordinaten, ohne Abweichung in Maßstab oder Größe, wogegen das grafische System von seiner Natur her relativ ist.



Feste Koordinaten definieren den ganzen Bildschirm



Die Koordinaten sind relativ zum Fenster,
der Maßstab wird durch den Wert y vorgegeben

Abb. 3.2 Koordinaten-Systeme

Der Basispunkt für das grafische System ist die untere linke Ecke. Doch es können auch mehrere grafische Koordinaten-Systeme, jedes mit seinem Basispunkt unten links in seinem eigenen Bereich, auf dem Bildschirm sein. Jedes dieser Koordinaten-Systeme ist ein Fenster. Die Idee dahinter ist, was immer passiert, welche Fenstergröße oder welcher Grafikmode auch benutzt wird: ein Kreis bleibt ein Kreis und ein Quadrat ein Quadrat.

Zur Erzielung dieser Unabhängigkeit darf man nicht die Pixel als Referenzpunkte benutzen, deshalb verwenden die Grafik-Prozeduren einen eigenen Maßstab zur Spezifikation der Größen. Der Befehl lautet:

```
SCALE Grösse-der-vertikalen-Achse
```

Der Aufbau eines Fensters für grafische Koordinaten, dessen Höhe 200 Einheiten beträgt, erfolgt mit dem Befehl

```
SCALE 200
```

Die Koordinaten für die Horizontale werden automatisch so gesetzt, daß Quadrate quadratisch sind und Kreise rund. Ein bequemer Weg zur Kontraktion und Expansion von Figuren ist also die Veränderung des SCALE-Faktors unter Beibehaltung der Koordinaten.

Interferenz-Muster

Für Anfänger der Computer-Grafik ist die Erzeugung von Interferenz-Mustern einer der verblüffendsten Effekte. Interferenz-Muster werden so genannt, weil sie den Interferenzen, die beim Studium von Wellenmustern auftreten, so ähnlich sind.

Wenn diese grafischen Muster auch wie Bilder von stehenden Wellen oder ähnliches aussehen, die Erklärung ist ziemlich kompliziert.

Beim Zeichnen einer geraden Linie (die in Wirklichkeit eine Reihe von kurzen Pixel-Linien ist) konnten wir sehen, daß zwei sich einander annähernde und sich kreuzende Linien am Kreuzungspunkt eine Interaktion verursachen. Kurze Pixel-Linien einer „großen" Linie können sich unter Umständen zu größere Pixel-Linien vereinigen.

Dieser Effekt ist leichter auf dem Bildschirm zu sehen als zu beschreiben: deshalb demonstriert die nachstehende kleine Prozedur den Effekt. Diese Prozedur unterstellt SCALE 100 und beginnt bei dem Punkt mit den grafischen Koordinaten 50,60. Es wird eine Reihe von Linien gezeichnet, die sich gegen den Uhrzeigersinn drehen, wobei der Winkel ständig um 2 Grad vergrößert wird.

```
DEFine PROCedure runde-interferenz-muster-1
   LOCAL winkel,x,y,d, radwinkel
   x=50:y=60:d=40
   FOR winkel=0 TO 358 STEP 2
     radwinkel=winkel * PI/180
     LINE x,y TO x-d *  SIN(radwinkel),y+d * COS(radwinkel)
   END FOR winkel
END DEFine
```

Der Effekt kann noch vergrößert werden, wenn man die Farben, in denen die Linien gezeichnet werden, wechselt. Zum Beispiel:

```
DEFine PROCedure runde-interferenz-muster-2
   LOCAL winkel,x,y,d,radwinkel
   x=50:y=60:d=40
   PAPER 0
   FOR winkel=0 TO 358 STEP 2
       INK 1+winkel MOD 7
       radwinkel=winkel * PI/180
       LINE x,y TO x-d * SIN(radwinkel),y+d * COS(radwinkel)
   END FOR
END DEFine
```

Hierdurch wird der Hintergrund auf Farbe 0 gesetzt (Schwarz bei 256-Pixel-Mode) und die Vordergrund-Linien zirkulieren durch die Farben 1 bis 7. Das Ergebnis von "winkel MOD 7" ist immer zwischen 0 und 6, und da „winkel" durch die Sequenz 0,2,4,6,8,10,12,14,usw. geht, folgt das Ergebnis von „winkel MOD 7" der Sequenz 0,2,4,6,1,3,5,0. Wenn man die Farben auf diese Art benutzt, lassen sich glanzvolle Effekte erzielen.

Fenster und Kanäle

Bei der Beschreibung des Grafik-Koordinaten-Systems wurde das Konzept des „Fensters" bereits vorgestellt. Ein Fenster ist ein Bereich des Bildschirms, in dem Ein- und Ausgaben konzentriert werden können. Ein derartiges Fenster kann als Mini-Bildschirm betrachtet werden und es ist ohne weiteres möglich, mehrere Fenster auf einem Bildschirm zu haben. Mit jedem Fenster ist ein „Kanal" verbunden.

Die Position eines Fensters wird mit Hilfe des Pixel-Koordinaten-Systems bestimmt (obwohl alle Grafik-Prozeduren ein spezielles „Fenster-Grafik-Koordinaten-System" benutzen). Das Befehls-Format ist:

```
WINDOW kanal-nummer, x-pixel, y-pixel, x-breit, y-tief
```

wobei „x-pixel, y-pixel" sich auf die Pixel-Koordinaten der oberen, linken Ecke des Fensters beziehen. Der Parameter „kanal-nummer" ist optional, zur Vermeidung von Fehlern ist es jedoch ratsam, mit diesem Parameter zu arbeiten (auch wenn ich später meinen eigenen Rat mißachte).

Unterstellen wir, daß wir im oberen Teil des Bildschirms ein Fenster haben wollen, unter dem noch Platz für ungefähr 6 Ausgabezeilen verbleibt. Da die übliche Größe der Zeichen beim 256-Pixel-Modus 10 Pixel hoch ist (s.a. Tabelle 3.2), scheint es vernünftig, das obere Fenster bis zum Pixel 199 gehen zu lassen, das untere Fenster beginnt dann bei Pixel 204 und reicht bis zum untersten Pixel (also 255).

Dem oberen Fenster geben wir die Kanalnummer 1, wogegen das untere den Standard-Kanal 0 bekommt. Die Standardfarbe für den Hintergrund ist schwarz, für das untere Fenster soll der Hintergrund blau sein.

```
PAPER 1:CLS
```

Die Farbcodes sind der Tabelle 3.3 zu entnehmen.

Beachten Sie, daß die Regeln der Farbaddition zu berücksichtigen sind, zur Erzeugung von gelb (Code 6 im 256-Modus) addiert man grün (Code 4) und rot (Code 2). Die Farbcodes korrespondieren im 256-Modus mit den auf den drei Grundfarben rot, grün und blau basierenden Farbkombinationen.
Die Initialen der Basisfarben sind RGB, deshalb nennt man einen Farbmo-

Tabelle 3.3 QL-Farbcodes

Code	256-Modus	512-Modus
0	schwarz	schwarz
1	blau	schwarz
2	rot	rot
3	violett	rot
4	grün	grün
5	oliv	grün
6	gelb	weiß
7	weiß	weiß

nitor, der über die drei Grundfarben verfügt und alle anderen Farben aus der Addition dieser Grundfarben bildet, einen RGB-Monitor.

Der Befehl PAPER 1, gefolgt von einem Clear Screen (CLS), produziert im 256-Modus einen blauen Hintergrund. Da gelbe Zeichen auf blauem Hintergrund sehr gut lesbar sind, können wir die o.a. Zeile entsprechend ändern:

```
PAPER 1: INK 6: CLS
```

Beachten Sie, daß die Ziffern Eins plus Sechs die Sieben ergeben und deshalb „Komplementärfarben" genannt werden, denn alle drei Grundfarben (blau, rot und grün) sind einmal in der Kombination enthalten.

Jetzt zur Definition des Fensters mit der Standard-Kanalnummer am unteren Ende des Bildschirms. Als Erstes sind die Pixel-Koordinaten zu fixieren. Wir haben die obere linke Koordinate bereits vorher auf 0,204 bestimmt (also x-pixel = 0 und y-pixel = 204). Der x-weit Parameter ist 512 (selbst wenn wir im 256-Modus arbeiten) und der y-tief Parameter ist 52 Pixel. Zur Definition des Fensters müssen wir also

```
WINDOW 0, 204, 512, 52
```

angeben, und dieser Befehl impliziert den Standard-Kanal. Jetzt müssen wir noch das obere Fenster definieren:

```
WINDOW 1, 0, 0, 512, 200
```

Hierdurch wird ein Fenster der vorgegebenen Dimensionen dem Kanal 1 zugeordnet. Da wir in dem oberen Fenster grafische Zeichnungen darstellen wollen, sollte der Hintergrund weiß sein und die Linien werden schwarz gezogen:

```
PAPER 1,0: INK 1, 7: CLS 1
```

Zur Erzielung eines mehrfenstrigen Grafik-Systems benötigen wir also folgende Befehle:

```
PAPER 1:INK 6:CLS
WINDOW 0, 204, 512, 52
WINDOW 1, 0, 0, 512, 200
PAPER 1, 0: INK 1, 7: CLS 1
```

Vor der Anwendung des (zum Beispiel) LINE-Befehls müssen wir die Startposition des Cursors bestimmen:

```
SCALE 800:  CURSOR 1,  400, 600
```

Danach sprechen alle Zeichnungs-Befehle automatisch das dem Kanal 1 zugeordnete Fenster an, beginnend bei der Cursor-Position.

Der Tongenerator

Der Ton wird auf dem QL durch den Intel Mikroprozessor generiert. Bis, wie versprochen, die weiteren Möglichkeiten zur Erzeugung von Tönen besprochen werden, muß der BEEP-Befehl ausreichen.

Dieser Befehl ist so komplex, daß man ihn durch Experimente erforschen muß. Immerhin entsprechen die ersten beiden Parameter denen des BEEP-Befehls auf dem Spectrum, darauf aufbauend hier ein BEEP-Programm:

```
REPeat  bruder-jack
        INPUT  "Wie lange?";X
        BEEP X,6 : BEEP X, 8 : BEEP X, 10 : BEEP X, 6
        BEEP X,6 : BEEP X, 8 : BEEP X, 10 : BEEP X, 6
```

```
        BEEP X,10: BEEP X,11 : BEEP 2*, X,13
        BEEP X,10: BEEP X,11 : BEEP 2*, X,13
        BEEP X/2,13 : BEEP X/2,15 : BEEP X, 10 : BEEP X,6
        BEEP X/2,13 : BEEP X/2,15 : BEEP X, 10 : BEEP X,6
        BEEP X,6 : BEEP X,1 : BEEP 2* X,6
        BEEP X,6 : BEEP X,1 : BEEP 2* X,6
END REPeat  bruder-jack
```

Mein Desinteresse für Ton und Musik auf Mikrocomputern ist groß und absolut.

4. Ein Turtle-Grafik System

Eine zunehmend populärere Methode zur Erzeugung komplexer grafischer Effekte ist die Turtle-Grafik.

Das Quadrat in Abbildung 4.1 wurde mit dem einfachen Befehl

```
square 300
```

erzeugt, und die beiden Quadrate in Abbildung 4.2 kommen durch die wesentlich komplexere Anweisung

```
square 300
rt 30
square 100
```

zustande. Hierbei ist es offensichtlich, daß der Befehl „square" die Anweisung zum Zeichnen eines Quadrates ist. Die Anweisung „square 300" bewirkt

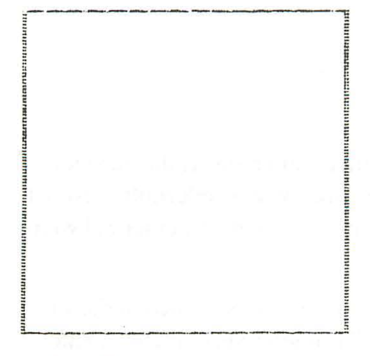




Abb. 4.1

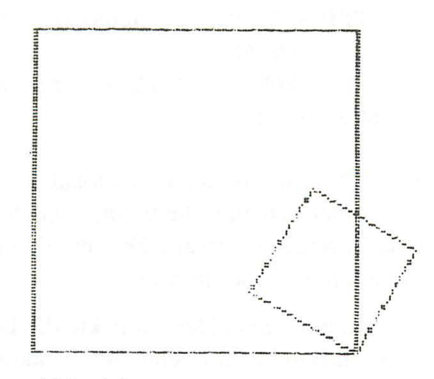


Abb. 4.2

das Zeichnen eines Quadrats mit der Seitenlänge 300. Wenn wir die Abbildung 4.2 untersuchen und das Diagramm mit den Befehlen vergleichen, stellen wir fest, daß das Kommando „rt 30“ bedeuten muß: Rechts drehen (turn right, oder etwas ähnliches), denn das Quadrat der Seitenlänge 100 wurde im Winkel von 30 Grad zu dem größeren Quadrat gezeichnet.

Irgend etwas muß sich drehen, zumindestens müssen wir uns etwas vorstellen, das sich dreht: Das „Ding“, das wir uns vorstellen, wird „Turtle“ (= Schildkröte) genannt. Unsere imaginäre Schildkröte zeichnet das Quadrat, nicht der Computer. Es ist die imaginäre Schildkröte, die mit einem Stift die Linien zieht.

Turtle Geometrie

Das nennenswerteste Buch über Turtle-Grafik ist Seymour Paperts schon am Anfang genanntes Buch „Mindstorms“ (Sie erinnern sich an Proteus?). Turtle-Grafik wurde zuerst bei LOGO als Bestandteil der Sprache eingeführt (zu LOGO verweise ich auf mein Buch: A pocket guide to LOGO). Turtle-Grafik und die Programmiersprache LOGO waren sehr erfolgreich in ihren Ansätzen: erstens, das Programmieren leicht erlernbar zu machen; zweitens, eine kräftige Sprache zu produzieren.

SuperBASIC ist eine kräftige Sprache und, mit der richtigen Einstellung angegangen, eine leicht erlernbare. So ist zum Beispiel zur Erzeugung des Turtle-Grafik-Befehls „square“ folgende Konstruktion ausreichend:

```
DEFine PROCedure square(seite)
      LOCAL i
      FOR i = 1 TO 4 : fd seite : lt 90
END DEFine
```

Diese Prozedur definiert den lokalen Bezeichner i und benutzt ihn zur Steuerung einer Schleife, die viermal durchlaufen wird. Was wiederholt wird, ist eine Bewegung vorwärts über die Distanz „seite“, gefolgt von einer Schwenkung um 90 Grad nach links.

Beachten Sie zwei Hauptaspekte der DEFinition: Erstens brauchen die Prozeduren/Befehle „fd“ und „lt“ die aktuellen Parameterwerte nicht in Klammern zu schließen (was auch für „square“ gilt); Zweitens, die FOR-Schleife

beansprucht nur eine logische Zeile, deshalb ist ein END FOR i oder ein NEXT i nicht nötig. Natürlich hätte man die Prozedur auch auf mindestens zwei Arten wesentlich umständlicher schreiben können:

```
DEFine PROCedure square(seite)
       LOCAL i
       FOR i = 1 TO 4
               fd seite
               lt 90
       END FOR i
END DEFine

DEFine PROCedure square(seite)
       LOCAL i
       FOR i = 1 TO 4
               fd seite
               lt 90
       NEXT i
END DEFine
```

Vom Konzept her ist der Gebrauch der „Schildkröte" sehr hilfreich. Ein Quadrat wird gezeichnet durch „vorwärts gehen" über die Distanz „seite", eine Drehung um 90 Grad, mit dreimaliger Wiederholung dieses Vorgangs. Die Simplizität ist zu bewundern.

Wir wollen jetzt untersuchen, welche Befehle für ein einigermaßen brauchbares Turtle-Grafik System benötigt werden. Zur Vereinfachung werden wir beim Aufbau des Systems soweit als möglich die gekürzten Formen der LOGO-Befehle benutzen. Das heißt, anstatt „vorwärts" werden wir „fd" benutzen. Außerdem werden wir Großbuchstaben statt Kleinschreibung anwenden (also FD statt fd) und versuchen damit, soviel als möglich im LOGO-Stil zu arbeiten. SuperBASIC-Turtle-Grafik kann genauso komplett sein wie bei LOGO, aber in diesem Kapitel wollen wir nur die wichtigsten Elemente des Systems betrachten.

Die wichtigsten Befehle für die Schildkröte

Die zu implementierenden Schlüssel-Befehle sind:

RT winkel	Rechtsdrehung um „winkel" Grad
LT winkel	Linksdrehung um „winkel" Grad
TT winkel	Drehe zum angegebenen „winkel" (in Grad)
FD distanz	Gehe vorwärts „distanz" Einheiten
BK distanz	Gehe rückwärts „distanz" Einheiten
GT	Gehe zu den Koordinaten „x,y"
CT	Zentriere die „Schildkröte"
PD	Ziehe Linie (mit PEN DOWN)
PU	Ziehe keine Linie (PEN UP)
GC	Lösche Grafik Areal/Fenster
TC	Lösche Text Areal/Fenster
TG	Initialisiere Turtle Grafik

Zur Kontrolle von Winkel, Position und Pen-Status der Schildkröte werden bestimmte, allgemeingültige Bezeichner benötigt:

XCOR	X-Koordinate der Schildkröte
YCOR	Y-Koordinate der Schildkröte
ANGLE	Winkel-Orientierung der Schildkröte
PSTATE	PEN DOWN ist 0, sonst ist PEN UP

Bei der Initialisierung des Turtle-Systems müssen allen diesen Bezeichnern Werte zugeordnet werden. Das zu benutzende Grafik-Koordinaten-System ist das im Kapitel 3 bei den Fenstern und Kanälen implizierte System.

Initialisierung der Turtle-Grafik

Im Kapitel 3 diskutierten wir die Erzeugung von zwei Fenstern im Bildschirm, wobei wir das obere Fenster für grafische Effekte und das untere für Text verwenden wollten. Bei der Bestimmung der Fenster haben wir den vertikalen Maßstab des oberen Fensters auf 800 gesetzt, woraus sich ein Maßstab von ungefähr 1200 für die Horizontale ergibt.

Am Ende der Betrachtungen stand die Zeile:

```
SCALE 800 : CURSOR 400, 600
```

Hierdurch wurde der vertikale Maßstab auf 800 Einheiten und der Grafik-Cursor fast genau in die Mitte des Fensters gesetzt. Bevor wir etwas anderes unternehmen, sollten wir die Prozedur CT (zentrieren des Cursors) definieren:

```
DEFine PROCedure  CT
       SCALE 800 : Cursor 400, 600
END DEFine
```

Die Prozedur zum Aufsetzen des Turtle-Grafik-Systems sieht jetzt wie folgt aus:

```
DEFine PROCedure  TG
       XCOR = 0 : YCOR = 0 : ANGLE = 0 : PSTATE = 0
       PAPER = 1 : INK = 6 : CLS
       WINDOW 0, 204, 512, 52
       TC
       WINDOW 1, 0, 0, 512, 200
       GC
END DEFine
```

Diese Prozedur bezieht sich auf die Prozeduren TC und GC zum Löschen des Textfensters (überflüssig) und des Grafikfensters. Hier sind die dafür benötigten Prozeduren:

```
DEFine PROCedure  TC
       PAPER 1 : INK 6 : CLS
END DEFine

DEFine PROCedure  GC
       PAPER 1, 0 : INK 1, 7 : CLS
END DEFine
```

Und das letzte Paar von Anwendungsroutinen ist:

```
DEFine PROCedure  PD
       PSTATE = 0
END DEFine

DEFine PROCedure  PU
       PSTATE = 1
END DEFine
```

Obwohl man die Anweisungen für den PenSTATE auch direkt geben kann (z.B.: PSTATE = 1), so wird doch die Programmierung durch die Anwendung von PD und PU wesentlich übersichtlicher.

Das Arbeiten mit Winkeln

Die Schildkröte startet immer aufwärts, und das entspricht dem Winkel von 0 Grad. Alle Winkel werden gegen den Uhrzeigersinn von 0 bis 359 Grad gemessen. Die Einhaltung des Bereichs von 0 bis 359 Grad wird durch die Benutzung der Funktion ANGNORM kontrolliert:

```
DEFine FuNction  ANGNORM (A)
       A = A-INT(A/360)*360
       IF A < 0 THEN A=A+360
       RETURN A
END DEFine
```

Diese Funktion konvertiert alle Winkelangaben, ob positiv oder negativ, in einen positiven Wert zwischen 0 und 359 Grad.

Zum Ziehen eines Winkels benutzen wir jetzt die Prozedur TT, die nachstehend definiert ist:

```
DEFine PROCedure TT (neuer-winkel)
       ANGLE = ANGNORM neuer-winkel
END DEFine
```

und die beiden anderen Winkel-Befehle (LT und RT) definieren wir:

```
DEFine PROCedure  LT(winkel-änderung)
       ANGLE = ANGNORM (ANGLE + winkel-änderung)
END DEFine

DEFine PROCedure  RT(winkel-änderung)
       ANGLE = ANGNORM (ANGLE - winkel-änderung)
END DEFine
```

und schon sind wir fertig mit der Manipulation der Winkel.

Bewegungs-Befehle

Bei der Konstruktion der runden Interferenz-Muster hat die Schildkröte in Wirklichkeit Linien bei sich ändernden Winkeln gezeichnet. Einen ähnlichen Satz von Gleichungen können wir für die Definition der Bewegungen benutzen. Zuerst jedoch wollen wir den leichten Befehl: „GO TO vorgegebenen Koordinaten" bauen.

```
DEFine PROCedure  GT(x-koord, y-koord)
       XCOR=x-koord:YCOR=y-koord
       IF PSTATE=1 THEN
                   MOVE XCOR, YCOR
       ELSE
                   LINE TO XCOR, YCOR
       END IF
END DEFine
```

Danach zeigt die Schildkröte wieder in dieselbe Richtung, in die sie vor dem Ziehen der Linie zeigte.

Auch das Kommando „vorwärts" (forward) läßt sich ohne Probleme erstellen.

```
DEFine PROCedure  FD(distanz)
       XCOR=XCOR-distanz * SIN(ANGLE * PI/180)
       YCOR=YCOR-distanz * COS(ANGLE * PI/180)
```

```
        IF PSTATE=1 THEN
                    MOVE XCOR, YCOR
        ELSE
                    LINE TO XCOR, YCOR
        END IF
END DEFine
```

Natürlich kann man GT und FD durch die Prozedur PLOT auch vereinfachen:

```
DEFine PROCedure PLOT
        IF PSTATE=1 THEN
                    MOVE XCOR, YCOR
        ELSE
                    LINE TO XCOR, YCOR
        END IF
END DEFine

DEFine PROCedure  GT(x-koord, y-koord)
        XCOR=x-koord:YCOR=y-koord
        PLOT
END DEFine

DEFine PROCedure  FD(distanz)
        XCOR=XCOR-distanz * SIN(ANGLE * PI/180)
        YCOR=YCOR+distanz * COS(ANGLE * PI/180)
        PLOT
END DEFine

DEFine PROCedure  BK(distanz)
        FD -distanz
END DEFine
```

Einige Beispiele

Die Grafiken in Abbildung 4.3 bis 4.5 sind Beispiele einer speziellen Kurvenart, die bei der Turtle-Grafik „Einwärts-Spirale" genannt wird. Erzeugt wird sie durch Variation der Parameterwerte:

```
DEFine PROCedure EINSPIRAL (seite,winkel,inkr)
       FD seite
       LT winkel
       EINSPIRAL seite,winkel+inkr, inkr
END DEFine
```

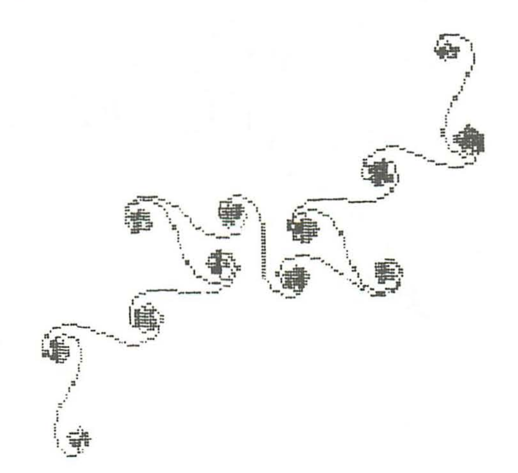


Abb. 4.3

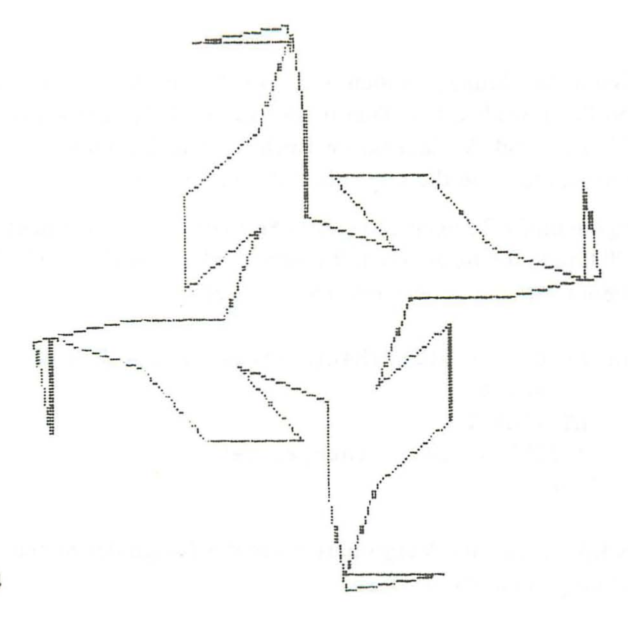


Abb. 4.4



Abb. 4.5

Die gezeigten Abbildungen haben ungefähr die gleiche Auflösung wie der QL im 256-Pixel-Modus. Die Theorie der Einwärts-Spirale wurde übrigens von H. Abelson und A. deSessa im Buch „Turtle-Geometrie“ ausführlich beschrieben, ebenso wie die folgende Art von Spirale.

Abbildung 4.6 und 4.7 zeigen „Auswärts-Spiralen“, die erste mit einem Winkel von 120 Grad und die andere mit einem Winkel von 117 Grad. Auswärts-Spiralen benutzen eine andere rekursive Prozedur:

```
DEFine PROCedure AUSSPIRAL(seite,winkel,inkr)
       FD seite
       LT winkel
       AUSSPIRAL seite+inkr,winkel,inkr
END DEFine
```

wobei das Inkrement (die Vergrößerung) in der Länge der Seiten und nicht im Winkel vorgenommen wird.

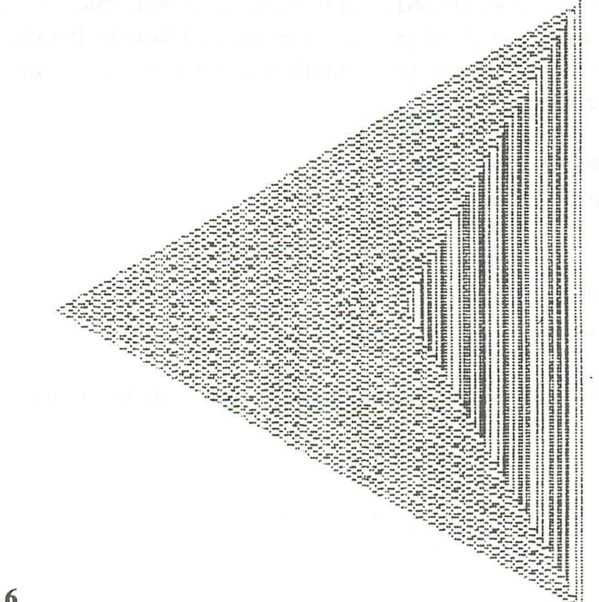


Abb. 4.6

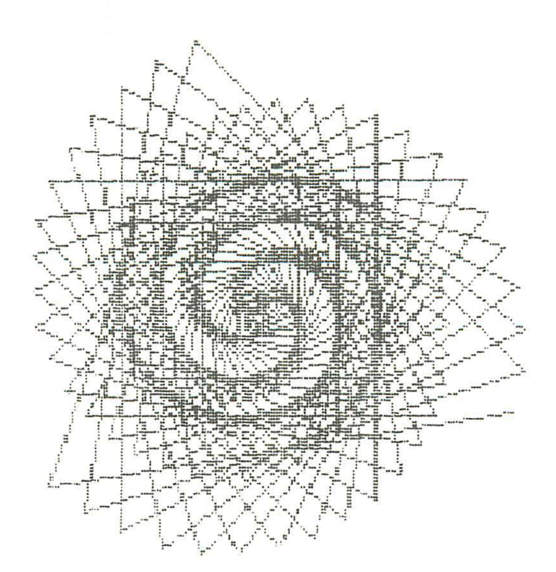


Abb. 4.7

Abschließend wollen wir den Kreis in Abbildung 4.8 betrachten. Dieser Kreis ist in Wirklichkeit ein dreißigseitiges Vieleck und könnte iterativ mit der FOR-Schleife oder rekursiv programmiert werden. Ich ziehe die endlose REPeat-Schleife vor.

```
DEFine PROCedure  KREIS (seite)
       REPeat endlos-schleife
              FD seite
              LT 12
       END REPeat
END DEFine
```

Vom Computer gezogene Kreise sind nichts weiter als vielseitige Figuren.

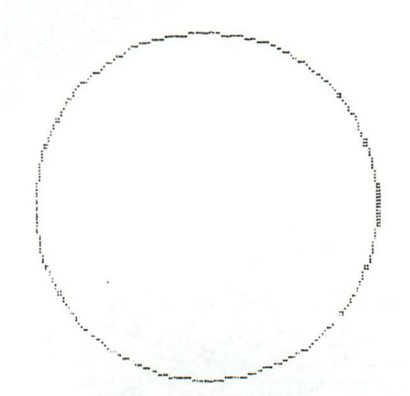


Abb. 4.8

5. Überlegungen zur SuperBASIC-Syntax

Die Syntax einer Sprache ist die Grammatik der Sprache. Die Syntax einer Programmiersprache ist die Gesamtheit aller zulässigen Anweisungen und Befehle, die in der Sprache verwendet werden können.

Obwohl es gewisse Ähnlichkeiten gibt, so ist SuperBASIC doch ganz anders als jedes andere BASIC. Im SuperBASIC gibt es bestimmte neue Grund-Ideen, deren Einfluß in der ganzen Sprache fühlbar ist. Diese Grund-Ideen sind:

Bezeichner, Paragraphen, Coercion, Tabellen und Slicing. Es gibt noch viele andere interessante Merkmale wie zum Beispiel *inter alia* (die Möglichkeit, zwischen logischen Operationen mit Binärwerten und wahren Werten zu wählen), Strings zu vergleichen, sowie die Idee der Fenster und Kanäle. Aber diese Möglichkeiten sind nicht so bedeutend wie die zuerst genannten.

Bezeichner

Die Idee der Bezeichner sowie die Idee der Paragraphen sind die beiden wichtigsten Aspekte in SuperBASIC.

Ein Bezeichner besteht aus einer Reihe gültiger Zeichen, die einen Namen bilden. Ein Bezeichner ist nicht das gleiche wie der Name einer Variablen. Mit jedem Bezeichner ist ein bestimmter „Modus" (um einen ALGOL 68 Terminus zu benutzen) oder ein bestimmter „Typ" (die PASCAL-Bezeichnung) verbunden. Wenn der SuperBASIC-Übersetzer einen Bezeichner findet, dann versucht er, den Modus des Items festzustellen, das durch den Bezeichner identifiziert wird.

Der Bezeichner wird dann mit einem Objekt im Computer verbunden und dieses Objekt hat diverse wichtige Attribute:

1. Es ist ein bestimmter Typ, z.B. eine Fließkomma-Variable oder ein String.
2. Es hat einen bestimmten Inhalt des vorgenannten Typs; als String zum Beispiel „abc“.
3. Es hat eine bestimmte, feste Adresse im Speicher.

Das wird deutlich sichtbar durch die Weitergabe von Parameter-Werten in Prozedur-Deklarationen. In SuperBASIC folgen die in Klammern stehenden „formalen“ Parameter dem Namen der Prozedur. Diese formalen Parameter sind zunächst lediglich Bezeichner (Namen) und unterliegen keinem bestimmten Modus oder Typ.

Die Natur des Objektes, auf das sich der Bezeichner bezieht, erkennt die Prozedur erst dann, wenn mit einem Prozedur-Aufruf die aktuellen Parameter-Werte geliefert werden. Wenn z. B. die aktuellen Parameter eine Tabelle identifizieren, dann behandelt die Prozedur den formalen Parameter-Bezeichner als Tabellen-Bezeichner.

Wenn der Bezeichner „diese-tabelle“ durch die Deklaration

DIM diese-tabelle(1,1)

als zweidimensionale Tabelle ausgewiesen wurde, dann ordnet das System aufgrund der Anweisung:

$$\text{diese-tabelle}=\left\{\{1,2\}\ \{3,4\}\right\}$$

dem Objekt (einer Tabelle), das durch den Bezeichner „diese-tabelle“ identifiziert wird, die entsprechenden Werte zu. Die Bedeutung des Konzepts der Bezeichner wird in den Abschnitten über Coercion und Slicing noch deutlicher.

Paragraphen

SuperBASIC arbeitet mit der Idee der klaren Bereichsabgrenzung der Programm-Kontrolle der verschiedenen Sprach-Konstruktionen. Bei Prozeduren, Funktionen, Repeat-Schleifen, For-Schleifen und der Select-Routine gehört immer eine abschließende „Klammer“ zur Routine. Diese Programm-

teile sind wie ein abgeschlossener Absatz (Paragraph) in einem Schriftstück, wie der in Klammern stehende Teil einer mathematischen Formel oder wie ein kompletter Satz.

In der gesprochenen Sprache ist auch der Satz ein in sich abgeschlossener Paragraph, er hat einen Anfang und ein klares Ende, den Punkt. Innerhalb eines Satzes können wieder andere, eventuell kleinere, Paragraphen sein, die zum Teil wiederum untergeordnete Paragraphen enthalten können. SuperBASIC scheint nach dem gleichen Prinzip zu arbeiten.

Ein Paragraph in SuperBASIC hat immer einen zugehörigen Bezeichner, der einen bestimmten Modus oder Typ hat. Er enthält eine Aussage, hier meistens eine Reihe von Anweisungen und auch andere Paragraphen, und der Bezeichner zeigt auf eine bestimmte Speicheradresse, an der die Informationen gespeichert sind.

Die Benutzung und die herausragende Bedeutung der Bezeichner führt dazu, daß die Reihenfolge der Paragraphen unwichtig ist. SuperBASIC ist eine funktionale (nicht so sehr sequentielle) Sprache. Bezeichner können Werte haben, die Operationen an anderen Bezeichnern beinhalten, und jeder Paragraph produziert ein Ergebnis, auch wenn es anstelle eines Wertes nur eine Aktion sein sollte. Wie ALGOL 68, LOGO und FORTH, oder wie LISP, ist auch SuperBASIC nichts weiter als eine Hierarchie von Operationen. Genauer gesagt hat SuperBASIC die Fähigkeit und Stärke, so zu arbeiten.

Durchschnittliche BASIC-Dialekte arbeiten praktisch nur sequentiell und es fehlt ihnen die Fähigkeit, funktional zu arbeiten (so gibt es zum Beispiel nur einen Parameter je Funktion in der Funktions-Beschreibung).
Die Programmiersprache PASCAL ist ebenfalls sequentiell und nicht funktional, das erklärt wahrscheinlich, warum sie nicht so erfolgreich ist, wie man gehofft hatte. In PASCAL akzeptiert das System einen Namen nur dann, wenn der zugehörige Typ bereits definiert ist. PASCAL verlangt eine strenge Ordnung, die eine elegante und kräftige Programmierung kaum ermöglicht.

Coercion

Coercion wurde schon vorher genannt und ist mit dem Konzept der Bezeichner eng verbunden.

In der Praxis bedeutet Coercion, daß der SuperBASIC-Übersetzer grundsätzlich etwas logisch Vernünftiges aus den Bezeichnern eines Statements zu machen versucht. Wo die Logik nicht ganz klar ist, scheint eine Hierarchie der Informations-Faktoren für die Coercion zu bestehen. Die Hierarchie ist:

1 Zuordnung: Die linke Seite der Zuordnung bestimmt den endgültigen Typ des Coercion-Ergebnisses;

2 Operand: Der Typ des Operanden auf der rechten Seite bestimmt den Modus des Objekts, auf das sich der Bezeichner bezieht;

3 Funktionen: Die Art der Funktionen (soweit vorhanden) bestimmt, was mit den aktuellen Parametern des Bezeichners geschieht;

4 Bezeichner: Der zu einem Bezeichner gehörende Modus bestimmt, ob Coercion vorgenommen werden kann;

und das entspricht der Hierarchie in ALGOL 68.

Folgendes ergibt sich daraus: Einige Anpassungen sind möglich, andere nicht, und ob eine Coercion möglich ist, ist letztlich vom Bezeichner abhängig. Diese Hierarchie der kontrollierenden Faktoren scheint gegensätzlich zu den Informations-Faktoren geordnet zu sein.

Slicing

Slicing kann in mancher Hinsicht als eine Art der Coercion verstanden werden.
Slicing ist eine saubere Technik zur Zuordnung von Werten zu Tabellen, bei gleichzeitiger Coercion der Werte, so daß sie passen. Die großen Vorteile dieser Technik müssen durch die praktische Arbeit bewiesen werden, aber in einigen allgemeinen Charakteristika erkennt man doch die obigen Ideen.

Unterstellen wir zwei Tabellen:

```
DIMension erste-tabelle(4,4,5)
DIMension zweite-tabelle(3,3,3)
```

Werte aus der ersten Tabelle können der zweiten Tabelle durch

```
zweite-tabelle=erste-tabelle(1 TO 4, 0 TO 3, 2 TO 5)
```

zugeordnet werden (immer unter Berücksichtigung der Tatsache, daß alle Tabellen beim Element 0 beginnen!). Ebenfalls gültig ist die Anweisung:

```
erste-tabelle(1 TO 4, 0 TO 3, 2 TO 5)=zweite-tabelle
```

Die Zuordnung unter Verwendung der Slicing-Methode ist eine Variante der Coercion. Das Slicing durch Anwendung der „niedriger TO höher“ Technik geht nicht ohne Coercion, und Coercion braucht die ausgefeilte Technik der Bezeichner.

6. Der 8049 Einchip-Computer

Ein Grund, warum der Sinclair QL so eine „aufregende“ Maschine ist, liegt darin, daß der QL den Motorola MC 68008 Mikroprozessor benutzt. Der MC 68008 Chip soll im Kapitel 7 untersucht werden, denn in diesem Kapitel wollen wir uns auf den weniger glamourösen Intel 8049 Chip konzentrieren, den der QL zur Kontrolle der Tastatur und anderer Geräte benutzt.

Der 8049 ist auf seine Art auch ein hochinteressanter Mikroprozessor, denn er ist ein kompletter 8-Bit-Mikroprozessor in einem Chip. Zugegeben, der 8049 ist kein besonders komplexer oder leistungsstarker Chip, aber er ist selbständig, denn er hat eine eigene „Rechenmaschine“ und einen eigenen Speicher (sowohl für Read-Only- als auch für Random-Zugriff).

Die Komponenten eines Computers

Abbildung 6.1 zeigt in schematischer Form die Basis-Komponenten eines Computers. Dieses Schema gilt grundsätzlich, nicht nur für Mikrocomputer.

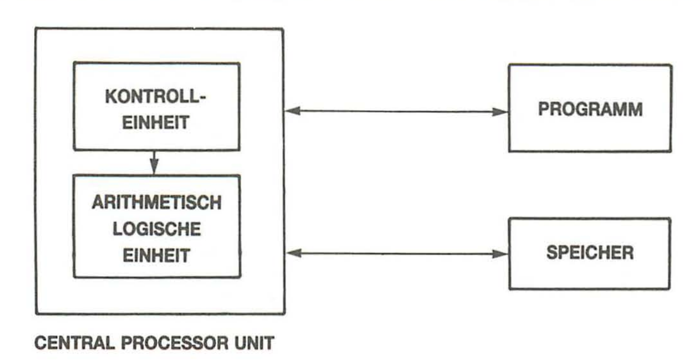


Abb. 6.1 Die Grundelemente eines Computers

Erstens: Es wird eine „Rechenmaschine“ benötigt, oder, wie es in der Fachsprache genannt wird, die „Arithmetisch Logische Einheit“ (ALU = arithmetic logic unit). In der ALU werden alle arithmetischen, logischen und ähnliche Operationen durchgeführt. Die ALU ist das Arbeitspferd, die Werkstatt. Wenn keine ALU vorhanden ist, dann kann nichts ausgeführt werden.

Zweitens: Die Rechenmaschine braucht Anweisungen, diese kommen von der „Kontroll-Einheit“ (control unit). Der Prozessor benötigt einen Bereich, der die Befehls-Codes für den Prozessor enthält und wo der Code in die aktuelle Operation für die ALU umgewandelt wird. Diese Einheit kontrolliert auch die Art der Kommunikation des Mikroprozessors mit anderen Komponenten des Computers, zum Beispiel mit dem Speicher oder den Eingabe- und Ausgabe-Ports. Die ALU und die Kontroll-Einheit zusammen werden vielfach als „Central Processor Unit“ (= Zentrale Prozessor Einheit) = CPU bezeichnet. Es ist zwar möglich, daß die ALU und die Kontroll-Einheit als physisch getrennte Einheiten existieren, eine derartige getrennte CPU ist bei Mikroprozessoren jedoch nicht üblich. Bei den früheren Computern war diese Trennung allerdings Standard.

Drittens: Es ist wenig sinnvoll, einen perfekt laufenden Motor zu haben, der unseren Wünschen gehorcht (zum Beispiel: im Leerlauf), wenn diese Maschine keine Leistung (z.B. ein Auto antreiben) vollbringen muß.

Zwischen den Instruktionen, die das Geschehen in der ALU steuern, und den Instruktionen, die die Leistung der CPU steuern, bestehen beträchtliche Unterschide.

Die Instruktionen, die angeben, welche Berechnungen durch die CPU vorgenommen werden sollen, nennt man im Allgemeinen „das Programm“. Auf der niedrigsten Ebene, und der 8049 Chip ist auf dieser Ebene, besteht das Programm aus einer Reihenfolge von Befehlen im Maschinencode, die direkt von der Kontroll-Einheit decodiert werden und die Operationen in der ALU auslösen.

Viertens: Es muß ein Element zum Speichern von Daten und Informationen vorhanden sein. Sowohl die im Programmablauf anfallenden Zwischen- und Endergebnisse, als auch sonstige Daten müssen gespei-

chert werden. Der Begriff „Speicher“ umschließt außer Daten- auch den Programmspeicher (s.a. Drittens).
Aus Gründen, die wir bald erkennen werden, sollte man zwischen diesen zwei Speicherarten unterscheiden.

Ein Einchip-Computer

Ein Einchip-Computer wird so genannt, weil ALU und Kontroll-Einheit (also die CPU) und der Programm- und der Datenspeicher in einer physischen Einheit zusammengefaßt sind, also in einem Chip.

Der eigentliche Chip ist ein hauchdünnes Silikon-Plättchen. Zum Schutz des Silikonscheibchens und um die Verbindung mit anderen Geräten zu ermöglichen, wird es in ein Plastik- oder Keramik-Gehäuse gepackt. Dieses gesamte Paket wird im allgemeinen „Chip“ genannt.
An den Seiten des kleinen Pakets sind Stifte angebracht (die elektrischen Verbinder), die in Schlitze der speziellen Haltevorrichtungen passen. Normalerweise sind die Stifte an zwei Seiten angebracht, deshalb nennt man diese Pakete DIPs (dual in-line packages). Es gibt allerdings auch Varianten mit Stiften auf allen Seiten. Der 8049 hat 40 Stifte und fällt in die Kategorie der „40 DIP“ Chips.

Die Grundstruktur des 8049 (man nennt es auch „Architektur“) ist in Abbildung 6.2 dargestellt. Es lohnt sich nicht, eine detaillierte Beschreibung aller Operationen des Prozessors zu geben, aber einige Charakteristika sind wichtig. Der Hauptpunkt ist dabei die Art, wie die vier Komponenten des Computers in dem Chip vorhanden sind.

Das erste Element eines Computers war die ALU und wie man der Abbildung 6.2 entnehmen kann, hat der 8049-Chip eine 8-Bit ALU und ein 8-Bit Akkumulatorregister. Eng verbunden mit der ALU ist das zweite Element, die Kontroll-Einheit. Auch wenn sie in dem Schema nicht gezeigt wird, so ist sie doch ein Bestandteil des Chips.

Das Programm und der ROM

Das dritte in einem Computer benötigte Element ist das Programm. Es ist in einem Baustein enthalten, der ROM genannt wird. ROM ist eine Abkürzung für Read Only Memory, also ein Speicher, der nur gelesen werden kann.

Der ROM ist also eine Anzahl von Speicherstellen, deren Inhalt festliegt und nicht geändert werden kann. Beim Computer wird ein ROM hauptsächlich zur Speicherung der Programme benutzt, die zum Betrieb des Systems nötig sind oder die Anwenderprogramme (zum Beispiel von BASIC) in die Maschinensprache übersetzen.

Beim QL steckt das Superprogramm für BASIC in einem ROM. Dieses „Superprogramm", meistens wird es Interpreter oder Übersetzer genannt, nimmt die vom Benutzer in den Datenspeicher eingegebene Information (d.h. BASIC-Befehle) und übersetzt diese Anwender-Befehle in Instruktionen, die die CPU versteht und ausführen kann.

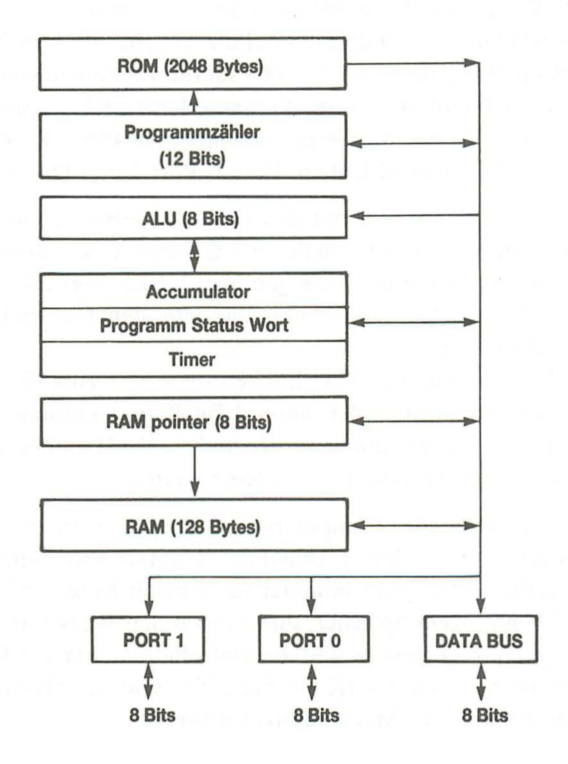


Abb. 6.2 Der Intel 8049, Grundstruktur

Es ist allgemein üblich, nicht nur beim QL, den BASIC-Übersetzer eines Mikrocomputers permanent in einem ROM zu speichern, da er dort nicht versehentlich zerstört werden kann. Bei manchen Computern muß der Übersetzer für andere Sprachen als BASIC in ein leicht zu veränderndes RAM geladen werden, da es keine Möglichkeiten gibt, zusätzliche ROM-Programme zu implementieren. Die Erläuterungen zum RAM erfolgten beim vierten Element des Computers.

Das Programm für den 8049 ist in einem ROM gespeichert, der eine Kapazität von 2 K-Bytes hat und sich mit im Chip befindet. Dieses Programm für den 8049 ist beim QL ein Satz von Befehlen, die diverse Ein- und Ausgabegeräte steuern.

Die Steuerung von Input und Output (kurz I/O=E/A=Ein-/Ausgabe) liegt demzufolge für den QL beim 8049 und der Hauptprozessor (MC 68008) des QL kann sich auf die Programme des Benutzers konzentrieren. Natürlich muß gelegentlich ein Programm im MC 68008 auf Informationen vom 8049 warten, besonders dann, wenn die Information von der Tastatur kommt.
Es gibt drei gute Gründe für die Benutzung des 8049 zur Kontrolle der I/O-Geräte. Diese drei Gründe heißen: Port 1, Port 2 und Daten-Bus.

Die I/O-Ports können direkt mit den Geräten verbunden werden und der Daten-Bus kann über Speicherstellen mit Geräten kommunizieren. Die beiden Ports und der Daten-Bus verfügen über 8 Bits. Port 2 kann auch (über vier der acht Bits) ein Gerät ansteuern, das unter dem Namen Intel 8243 I/O-Expander bekannt ist.
Durch die Verwendung des 8243 können diese vier Bits vier verschiedene 4-Bit I/O-Ports erzeugen. Jeder dieser 4-Bit Ports kann nun wiederum mit einem 8243 Expander verbunden werden und die 8243 können „kaskadieren", es können also beliebig viele Ports erzeugt werden.

Bei vielen konventionellen Computern muß der gleiche Prozessor sowohl die Ein- und Ausgabe als auch den Ablauf der Benutzer-Programme steuern (so zum Beispiel die vorherigen Computer von Sinclair Research). Die Notwendigkeit, bei begrenztem Speicherraum sowohl den Bildschirm als auch das BASIC-Programm zu steuern und zu kontrollieren, war der Grund für die geringe Geschwindigkeit des 1K Sinclair ZX81 und sie erklärt auch, warum der Bildschirm im FAST-Modus dunkel wurde.

Die Speicherung von Daten

Das vierte Element des Computers war die Speicherung von Daten. Der 8049-Chip verfügt über 128 Byte RAM (das ist 1/8 Kilobytes). RAM steht für „Random Access Memory“ (= Speicher mit willkürlichem Zugriff) und ist eine Speicherart, bei der Daten geschrieben und verändert werden können. Wenn der Prozessor angeschaltet (oder initialisiert) wird, stehen sofort Instruktionen aus dem ROM zur Verfügung. Zu diesem Zeitpunkt ist der Inhalt der RAM-Adressen noch nicht definiert.
Ein 128-Byte-Speicher ist nicht viel, aber für die Anwendungen, für die der 8049 konstruiert wurde, ist die Kapazität des RAM weniger wichtig als der dem Programm im ROM zur Verfügung stehende Speicherraum. Die Art, wie der RAM-Speicher vom 8049 benutzt wird, ist sehr interessant.

Der 8049 hat einige spezielle Speicherstellen, „Register“ genannt, für die es spezielle Operationen zur Datenmanipulation gibt. Bei vielen Mikroprozessoren ist der RAM-Speicher so klein, daß er völlig von den Registern verbraucht wird. Deshalb wird der Registerspeicher dann nicht als „on-chip-RAM“ deklariert, sondern lediglich als „Register“. Da der 8049 Chip einen größeren RAM-Speicher hat, ist es offensichtlich, daß die Register ein Teil des „on-chip-RAM“ sind.

Die ersten acht Speicherstellen im RAM werden für eine „Bank“ von 8 Registern verwendet, anschließend liegt ein „Stack“-Bereich (auch Stapelspeicher genannt), und dann kommen noch einmal acht Speicherstellen für Register. Der 8049 hat, wie auch der in den bisherigen ZX-Computern verwendete Zilog Z80, eine doppelte Reihe von Registern, zwischen denen man hin- und herschalten kann.

Im Anschluß an die zweite Reihe von Registern liegt der Benutzer-RAM, das sind Speicherstellen, die nicht für spezielle Funktionen vorgesehen sind. Normalerweise ist der gesamte RAM eines Mikroprozessors speziellen Funktionen (z.B. Registern) zugeordnet. Im 8049 können jedoch freie Kapazitäten im RAM für die Speicherung von Daten, die im ROM anfallen, benutzt werden.

Der Befehlsvorrat für den 8049 ist in gewisser Beziehung recht primitiv, da er für den geringen Programm-Speicherraum (im ROM) entworfen wurde. So ist zum Beispiel der Speicherraum in „Pages“ von 256 Bytes organisiert. Um nun von einer Page zu einer anderen zu verzweigen, braucht man eine

spezielle „Jump"-Instruktion, selbst wenn die Speicherstellen direkt aufeinander folgen.

Page 0, die erste Page, umfaßt die Speicherstellen 0 - 255. Wenn nun das Programm das Ende dieser Page erreicht hat, muß der letzte Befehl ein „Jump" sein, da das Programm nicht von der Speicherstelle 255 auf die Speicherstelle 256 weiterschalten kann.

Wenn RAM und ROM auch nicht sehr groß sind, eine Speichererweiterung durch Hinzuschaltung von externen ROMs oder RAMs ist ohne weiteres möglich, obwohl das zwangsläufig eine Reduzierung der Geschwindigkeit mit sich bringt. Je mehr der Speicher erweitert wird, desto mehr Jump-Befehle sind erforderlich.

Da der Programmzähler nur zwölf Bit umfaßt, kann der ROM-Speicher maximal 4 KByte Kapazität haben. Direkt auf dem Chip (on-chip) hat der 8049 bereits 2 K, so daß höchstens noch 2 K externer Speicher zugeschaltet werden können. Der 8049 betrachtet den ROM als zwei separate Bausteine mit je 2 K. Um nun den externen ROM zu benutzen, müssen die Register umgeschaltet werden.

7. Der MC 68008: Ein strukturierter Chip

Sinclair Research hat behauptet, daß der QL einen 32-Bit Mikroprozessor benutzt. Motorola Semiconductors spricht davon, daß die MC 68000 Serie 16-Bit Mikroprozessoren sind. Und der Motorola MC 68008, der Prozessor im QL, holt die Informationen Byte für Byte aus dem Speicher (genau wie andere Mikroprozessoren, z.B. der MOS Technology MCS 6502 oder der Zilog Z80).

Wenn Sinclair Research den MC 68008 als 32-Bit Mikroprozessor deklariert, dann beziehen sie sich hauptsächlich auf die interne Struktur des Chips. Der MC 68008 hat (wie auch der schnellere MC 68000) einen großen und bedeutenden Satz von siebzehn 32-Bit-Registern, und diese Register sind im „on-chip-RAM" (s.a. Kapitel 6).
Diese 32-Bit Register im MC 68008 können mit den vier 8-Bit Registern des MCS 6502 verglichen werden oder mit dem größeren Satz von sieben 8-Bit Registern im Zilog Z80, bei dem 6 Register so kombiniert werden können, daß sich drei Pseudo-Register je 16 Bit ergeben. Der Z80 ist insofern bemerkenswert, als er (wie auch der Intel 8049) zwei Bänke austauschbarer Register hat.

Der einzige 8-Bit Mikroprozessor mit echten 16-Bit Registern ist der überlegene Motorola MC 6809, der über 5 echte 16-Bit Register verfügt und zwei spezielle 8-Bit Register, Akkumulatoren genannt, hat (sie können kombiniert werden und ergeben so ein 16-Bit Register).

Der Adress-Bus

Wie bereits im Kapitel 6 festgestellt wurde, braucht jeder Mikroprozessor die Möglichkeit, mit seiner Umgebung zu kommunizieren. Diese Umgebung

besteht nun in der Hauptsache aus dem Programm (ROM) und gespeicherten Daten (RAM), deshalb benutzt der MC 68008 eine Sprache, die aus Speicheradressen und dem Inhalt dieser Adressen besteht.

Ein Mikroprozessor „zeigt“ auf eine Speicherstelle durch die Angabe der Adresse. Die Ansprache einer Adresse erfolgt meistens über den Adress-Bus, jedoch hat der i8049 Chip keinen Adress-Bus, da sich der Speicher im Chip befindet. Für Adressen außerhalb des Chips (in externen Speichern) benötigt auch der 8049 einen Adress-Bus.
Sowohl der MC 6502 als auch der Z80 haben zusätzlich zu den normalen ein spezielles Register, das auch als Programmzähler bezeichnet wird. Bei beiden Mikroprozessoren umfaßt dieser Programmzähler 16 Bits.
Der Programmzähler enthält die Adresse der Speicherstelle, in der der auszuführende Befehl steht. Eine Binärzahl mit 16 Bits kann Werte von 0 bis 65535 annehmen, woraus sich 64 K verschiedene mögliche Adressen ergeben. Hierin liegt der Grund, warum die meisten 8-Bit Computer nur maximal 64 K Speicher haben (auf jeden Fall, solange keine speziellen Techniken benutzt werden).

Jedes Bit der Zahl im Programmzähler entspricht einer elektrischen Verbindung zwischen dem Mikroprozessor und dem Speicher des Computers (ROM und RAM). Jede elektrische Verbindung (in dem sogenannten Adress-Bus) entspricht einem Stift am Mikroprozessor-Baustein (der Einheit, in der das Silikon-Scheibchen ruht). Die Spannung an den Stiften des Adress-Busses ist entweder „high“ oder „low“, je nachdem, ob der entsprechende Bit-Wert im Programmzähler „eins“ oder „null“ ist.

Die sechzehn Stifte am MC 6502 und am Z80 werden also als eine Einheit betrachtet (sie bilden den Adress-Bus). Die sechzehn Leitungen des Adress-Busses werden durch eine Speicherzugriffs-Kontrolleinheit, meistens ein anderer Chip, so konvertiert, daß sie auf bestimmte Speicher-Bytes zeigen.

Der Daten-Bus

Datentransfer vom Speicher zum Mikroprozessor erfolgt immer dann, wenn an weiteren Stiften am Chip entsprechende Spannungen anliegen. Dabei kann die Information sowohl zu einer Speicherstelle fließen, als auch von dort empfangen werden. Demzufolge wird ein weiterer Stift benötigt, mit dem der Pro-

zessor die Richtung steuert. Ein Datenbyte umfaßt wie beim MC 6502 und dem Z80 8 Bit, was weiteren 8 Stiften am Chip entspricht.

Der Programmzähler des MC 68000 hat, wie auch alle anderen Register, zweiunddreißig Bits, wovon nur dreiundzwanzig für den Adresszähler benutzt werden. Der MC 68000 arbeitet in Einheiten von zwei Bytes gleichzeitig. Er hat also einen 16-Bit Datenbus und deshalb zwei weitere Stifte, die angeben, welches der beiden Bytes benötigt wird.
Es gibt also fünfundzwanzig Stifte zur Adressierung der Bytes, obwohl das Ergebnis dieser Adressierung nur einem 24-Bit Adress-Umfang entspricht. Der MC 68000 kann also 16 MByte, genau: 16.777.216 Byte, adressieren.

Der MC 68000 transferiert Daten in Paketen von sechzehn Bits, deshalb spricht man von einem 16-Bit Chip, obwohl er 32-Bit Register hat. Die Basis-Datentypen, mit denen er arbeiten kann sind: einzelne Bits, Bytes, Worte (= 2 Bytes) und lange Worte (= 2 Worte).

Der Mikroprozessor im QL ist aber kein MC 68000. Der QL benutzt eine modifizierte Version des MC 68000, den MC 68008 Mikroprozessor. Für den Assembler-Programmierer sind die beiden Prozessoren identisch (beachten Sie aber trotzdem Kapitel 8). In den meisten Fällen wird ein Maschinen-Code-Programm für den MC 68000 auch auf einem MC 68008 einwandfrei laufen. Es gibt aber Unterschiede; worin bestehen sie also?

Der wesentlichste und offensichtliche Unterschied zwischen dem MC 68008 und dem MC 68000 liegt darin, daß der MC 68008 einen 8-Bit Daten-Bus hat, wie die meisten 8-Bit Mikroprozessoren. Aber, im Gegensatz zu den 8-Bit Chips, die nur ein Byte transferieren können, arbeitet der MC 68008 mit langen Worten (wenn auch nur Byte für Byte). Die einzigen Datentypen für 8-Bit Chips sind Bits und Bytes.
Durch die Reduktion auf einen 8-Bit Datenbus konnte auch die Anzahl der benötigten Stifte um acht reduziert werden, wodurch der Prozessor-Baustein (der Chip) vereinfacht wurde. Viele andere System-Eigenschaften wurden ebenfalls vereinfacht, so kann man z.B. Standard 8-Bit Zusatz-Chips verwenden. Der MC 68008 ist zwar eine langsame Version des MC 68000 (jedoch erheblich schneller als normale 8-Bit Chips), aber er ist in Wirklichkeit ein 16-Bit Chip.

Ein anderer wichtiger Unterschied zwischen dem MC 68008 und dem MC 68000 ist die Tatsache, daß der Adress-Bus des MC 68008 zwanzig Bits umfaßt. Da das vier Bits weniger sind als beim Adress-Bus des MC 68000,

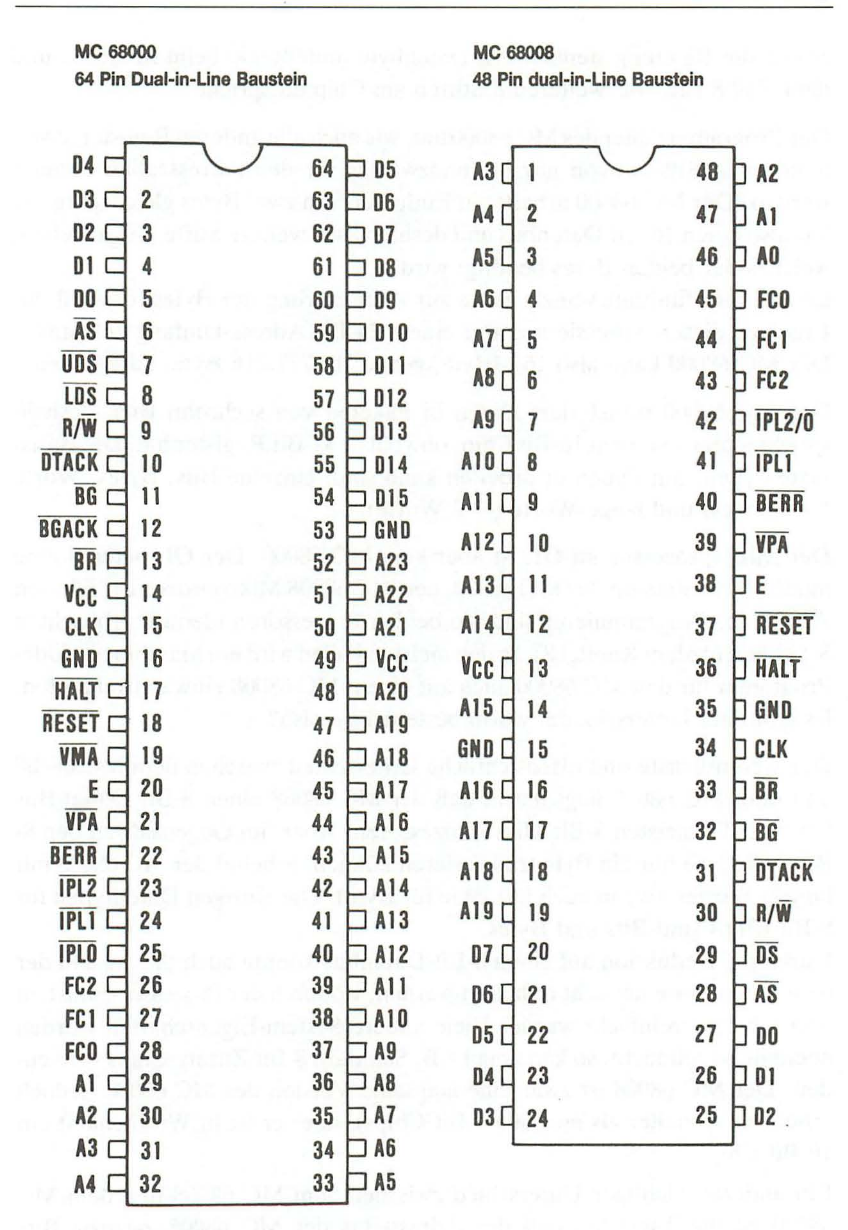


Abb. 7.1 Die Steckerbelegung

kann auch nur weniger Speicherraum adressiert werden. Im Gegensatz zum MC 68000 mit seinem Adress-Umfang von 16 Megabyte kann der MC 68008 nur 1 Megabyte adressieren.
Der Adress-Bereich des MC 68008 umfaßt also nur ein Sechzehntel der Möglichkeiten des MC 68000; aber man sollte nicht vergessen, daß ein Megabyte das sechzehnfache der 64 K ist, über die die meisten 8-Bit Prozessoren verfügen.

Der potentielle Adress-Bereich des MC 68008 ist also erheblich größer als bei den meisten anderen 8-Bit Chips. Beim QL ist jedoch nicht das ganze Potential erschlossen, denn die äußerste Grenze für Speichererweiterungen liegt bei 640 K. Irgendwann wird aber wohl ein Bastler oder eine Firma mit ausreichender Phantasie das Maximum von 640 K auf 1 Megabyte erhöhen.

Das MC 68008 Paket

Durch die Einsparung von fünf Stiften beim Adress-Bus, den eingesparten 8 Bit beim Daten-Bus und zusammen mit diversen anderen Vereinfachungen konnte beim MC 68008 die Anzahl der Stifte gegenüber dem MC 68000 von vierundsechzig auf achtundvierzig reduziert werden.

Abbildung 7.1 zeigt eine Gegenüberstellung der Steckerbelegung beim MC 68000 und beim MC 68008, beide dargestellt als DIP (= Dual-in-line = parallel). Die Erklärung der Bedeutung der meisten Anschlüsse wäre sehr technisch, aber einige Punkte sind doch erwähnenswert.

Der erste Punkt betrifft die Norm für die Darstellung der Bedeutung der Stifte, ob mit niedriger (low) oder hoher (high) Spannung gearbeitet wird. Betrachten wir Stift 9 beim MC 68000, der dem Stift 30 beim MC 68008 entspricht. Dieser Anschluß steuert das Schreiben und Lesen über den Daten-Bus. Die Linie über dem $\overline{W}$ bei R/$\overline{W}$ gibt an, daß bei niedriger Spannung ein Schreib-Zyklus über den Daten-Bus läuft. Diese Kennzeichnung gilt natürlich auch für alle anderen Stifte. $\overline{\text{RESET}}$ bedeutet also, daß ein System Reset (= die Reinitialisierung eines bestimmten Systemzustandes) erfolgt, wenn an diesem Stift eine niedrige Spannung anliegt. Bei einem vorübergehenden Spannungsabfall wird das System also reaktiviert, da die Spannung auch an diesem Stecker „low" ist.

Die Stifte A0 bis A19 beim MC 68008 bilden den Adress-Bus und jeder Stift entspricht einem Bit (also Bit 0 bis Bit 19) im Adressregister.

Beim MC 68000 bilden die Stecker A1 bis A 23 den Adress-Bus und gehören zu den Bits 1 bis 23 des Adress-Registers, wobei Bit 0 ignoriert wird. Allerdings wird es nicht ganz übersehen, denn die Stifte 7 und 8 ($\overline{UDS}$ und $\overline{LDS}$) informieren den Adress-Bus, ob die oberen und/oder die unteren Bytes zu holen sind.

Der Daten-Bus des MC 68008 wird von den Stiften D0 bis D7 gebildet, beim MC 68000 sind es die Stifte D0 bis D15. Die Bedeutung der niedrigen und höheren Spannung wird am deutlichsten, wenn man bedenkt, wie das R/$\overline{W}$ Signal mit dem $\overline{DS}$ Signal (Datenstrobe) zusammenarbeitet, um den MC 68008 über den Status des Daten-Busses zu informieren (s.a. Tabelle 7.1).

Tabelle 7.1 Steuerung des Daten-Busses durch Datenstrobe

$\overline{DS}$	R/$\overline{W}$	Datenbus (D0 - D7)
High	High/Low	Keine gültigen Daten
Low	High	Gültige Daten (Lesen)
Low	Low	Gültige Daten (Schreiben)

Falls ein Spannungsabfall eintritt (wenn also die Spannung an allen Stiften niedrig ist), schaltet der Mikroprozessor in den Status „Gültige Daten (schreiben)“. Hier liegt die Ursache dafür, daß nach einer derartigen Panne, z.B.: Schwankung in der Netzspannung, zufällige Inhalte im Speicher stehen können.
Das bedeutet jedoch nicht, daß derartige Zufallsinhalte auch in den Registern des Mikroprozessors landen können. Mit Zufallswerten im Speicher kann ein System „fertig“ werden, es ist aber wesentlich problematischer, mit einem Satz verfälschter Register zu arbeiten.

Interrupts und Ausnahmen

Beim MC 68000 sind die Stecker 23 bis 25 mit $\overline{IPL2}$, $\overline{IPL1}$ und $\overline{IPL0}$ markiert, hierbei handelt es sich um „Interrupt-Leitungen“ (= Unterbrechungs-Verbindungen). Ein Interrupt ist ein Signal zum Mikroprozessor, das von einem Peripheriegerät oder einem Fehler-Warnsystem ausgesandt wird.

Die MC 68000 Serie macht umfangreichen Gebrauch von den Interrupts, denn sie sind wesentlich effizienter als das auch übliche System zur Kommunikation mit der Peripherie und zur Fehlerbehandlung, dem „Polling". Wenn ein Computer das Polling benutzt, dann „fragt" er die Geräte der Reihe nach, ob sie mit dem Prozessor kommunizieren wollen. Eine derartige „rund um"-Abfrage nimmt viel Zeit in Anspruch.
Beim Interrupt-System arbeitet der Prozessor ungestört, bis er ein Interrupt-Signal bekommt: zum Beispiel von der Tastatur, die ein Zeichen senden oder einen Puffer entleeren will.

Es gibt verschiedene Interrupt-Ebenen. So haben zum Beispiel einige Interrupts Vorrang vor jeder anderen Prozessor-Arbeit, und andere Interrupts können ignoriert werden. Ein Interrupt, der jede andere Prozessor-Arbeit stoppt und nicht übergangen werden kann, heißt bei vielen Mikroprozessoren „unverdeckbarer Interrupt". Ein Interrupt, der (wenn nötig) übergangen werden kann, wird dagegen „verdeckbarer" Interrupt genannt.

Die MC 68000 Serie hat acht verschiedene Ebenen der Interrupt-Priorität, und auf dem MC 68000 selbst können alle acht Ebenen (von 0 bis 7) für die Kommunikation mit Peripherie und Speicher benutzt werden. Beim MC 68008 gibt es nur vier der Ebenen (level 0,2,5,7), weil die $\overline{IPL0}$- und $\overline{IPL2}$-Verbindungen des MC 68000 beim MC 68008 über einen Stift gehen (vgl. Stecker 41 und 42). Die Konsequenzen des gemeinsamen Steckers zeigt die Tabelle 7.2, aus der ersichtlich ist, daß der MC 68008 ein Interrupt-Signal nur dann richtig erkennen kann, wenn $\overline{IPL2}$ und $\overline{IPL0}$ das gleiche Potential haben. Das „xxx" in der Tabelle zeigt eine für den MC 68008 unverständliche Kombination.

Tabelle 7.2 Interrupt-Ebenen für den MC 68008

$\overline{IPL2}$	$\overline{IPL1}$	$\overline{IPL0}$	MC 68000		MC 68008	
High	High	High	000	0	000	0
High	High	Low	001	1	xxx	x
High	Low	High	010	2	010	2
High	Low	Low	011	3	xxx	x
Low	High	High	100	4	xxx	x
Low	High	Low	101	5	101	5
Low	Low	High	110	6	xxx	x
Low	Low	Low	111	7	111	7

Man kann deutlich erkennen, daß die vier Interrupt-Ebenen des MC 68008 auf den Ebenen 0, 2, 5 und 7 liegen. Level 0 wird als Anzeiger dafür benutzt, daß keine Interrupt-Anforderungen vorliegen (alle haben „high"-Spannung). Level 7 ist ein unverdeckbarer Interrupt (alle haben „low"-Spannung). Ein Spannungsabfall erzeugt also einen unverdeckbaren Interrupt, denn alle Spannungen sind „low". Es entsteht also ein Level 7 Interrupt.

Interrupts können eine Reihe von Ursachen haben, die von Motorola als „Ausnahmen" bezeichnet werden. Sie schließen Adressfehler, Übertretungen des Vorrechts, unzulässige Befehle, Bus-Fehler und das Reset-Signal ein.

Die Privelegien-Statusse

Der MC 68008 hat eine gut durchdachte Prozessor-Struktur, und das Konzept der Privilegien ist ein bedeutender Aspekt dieser Struktur. Der Prozessor kennt zwei Privileg-Statusse, den Benutzer-Status und den Supervisor-Status.

Der Privileg-Status bestimmt, welche Operationen zulässig sind. So gibt der Privileg-Status zum Beispiel an, wie externe Speicher-Management-Geräte zur Kontrolle des Datensendens und -Empfangens und zum Übersetzen zu benutzen sind. Außerdem zeigt der Privileg-Status an, ob bestimmte Befehle den „Supervisor-Stack-Pointer" oder den „Benutzer-Stack-Pointer" ansprechen.

Durch das Privileg-System kann die Sicherheit im Computer verbessert werden. So sollten zum Beispiel Programme nur auf ihre eigenen Codes und Daten zugreifen können, der Zugriff auf geschützte Informationen sollte unmöglich sein.
Ein Privileg-System ist besonders wichtig bei Systemen mit Multi-Tasking, wofür der QL ja gedacht ist. Wenn mehrere Programme voneinander unabhängig gleichzeitig laufen, dürfen sie sich nicht gegenseitig beeinflussen, man kann ihnen höchstens einen gemeinsamen Datenbereich zur Verfügung stellen.

Im Benutzer-Status, dem Normalzustand, werden die Programm-Zugriffe kontrolliert und die Auswirkungen auf andere Teile des Systems werden dadurch verringert. Die meisten Programme laufen im Benutzer-Status. Das Betriebssystem (z.B.: QDOS) arbeitet jedoch im Supervisor-Status, hat Zugriff zu allen Ressourcen und bewirkt die Zuteilung der Ressourcen für die Programme im Benutzer-Status.

Der Supervisor-Status beinhaltet das höhere Privileg. Der Status wird durch das S-Bit im Status-Register bestimmt. Wenn das S-Bit an ist (High) oder das System die Konsequenzen einer „Ausnahme“ (Interrupt) aufarbeitet, dann ist der Prozessor im Supervisor-Status. Alle MC 68000-Befehle können im Supervisor-Status ausgeführt werden; als Stack-Pointer wird dann der System-Stack-Pointer benutzt.
Der Benutzer-Status hat niedrigeren Rang und wird nur vom S-Bit gesteuert (das Bit ist „aus“ oder Low). Anstelle des System-Stack-Pointers wird der Benutzer-Stack-Pointer angesprochen und auch die meisten Maschinensprachen-Befehle können ausgeführt werden. Bestimmte Befehle sind im Benutzer-Status allerdings nicht ausführbar (= unzulässig): die STOP und RESET Befehle z.B. oder alle Befehle, die den System-Stack-Pointer verändern.

Zum Privileg-Status und zur Aufarbeitung von „Ausnahmen“ (meistens Interrupts) gehören drei Stifte, an denen die Funktions-Codes für den Prozessor gesetzt werden. Sie heißen FC 2, FC 1 und FC 0 und sind die Stifte 43 bis 45 des MC 68008 Bausteins. Die von ihnen ausgegebenen Potentiale sind gültig, wenn der Adress-Bus benutzt wird (d.h. wenn $\overline{AS}$ = Adress-Strobe aktiviert ist).

Tabelle 7.3 Funktions-Code Ausgaben

FC2	FC1	FC0	Operations-Zyklus
Low	Low	Low	(undestiniert, reserviert)
Low	Low	High	Benutzer-Daten
Low	High	Low	Benutzer-Programm
Low	High	High	(undestiniert, reserviert)
High	Low	Low	(undestiniert, reserviert)
High	Low	High	Supervisor-Daten
High	High	Low	Supervisor-Programm
High	High	High	Interrupt-Bestätigung

Die höchste Priorität bei der Benutzung des Adress-Busses haben die unbedingten (unverdeckbaren) Interrupt-Anforderungen. Die nächste Priorität hat dann die Abarbeitung des Supervisor-Programms, gefolgt vom Senden oder Empfangen von Daten für das Supervisor-Programm.
Die niedrigste Priorität in der Warteschlange vor dem Adress-Bus hat der Datenzugriff durch Benutzer-Programme, eine etwas höhere Priorität hat die

Abarbeitung des Benutzer-Programms selber. Nur wenn es kein Gedränge am Adress-Bus gibt, werden die Daten-Anforderungen des Benutzers sofort ausgeführt.

Die FC-Codes haben außer dieser Ordnungsfunktion noch weitere Aufgaben, da sie die Art der Informationen klassifizieren. Die Code-Kombinationen erlauben eine externe Übersetzung von Adressen und differenzieren die verschiedenen Prozessor-Statusse.

Die elektrischen Anschlüsse einiger der Stifte sind so, daß die Potentiale entweder vom Prozessor selbst kommen können (zum Beispiel bei den Adress-Leitungen) oder sie können von außen kommen (z.B. Interrupt-Leitungen). Bei den Anschlüssen, die den Daten-Bus bilden, können die Potentiale als Prozessor-Output vom Prozessor oder als Prozessor-Input von außen kommen. Aufgrund der Fähigkeit, sowohl senden als auch empfangen zu können, wird der Daten-Bus auch als „Bi-Direktional" bezeichnet.

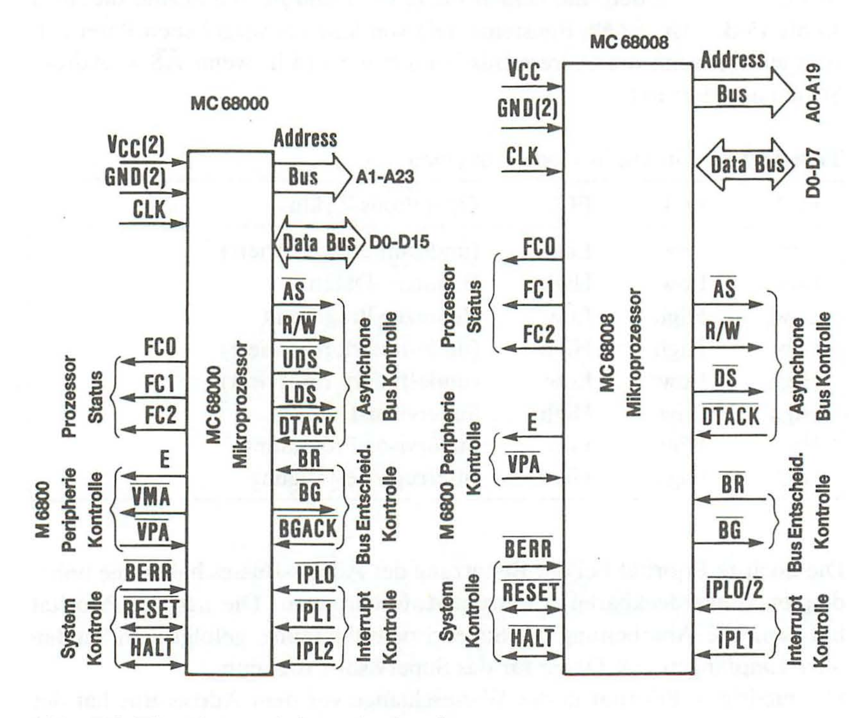


Abb. 7.2 Eingabe- und Ausgabesignale

Abbildung 7.2 zeigt eine Gegenüberstellung der funktionalen Gruppen der Input- und Outputsignale beim MC 68008 und beim MC 68000.

Die Architektur des MC 68000

Es wurde bereits erwähnt, daß die interne Architektur des MC 68008 der des leistungsfähigeren MC 68000 entspricht, deshalb wurde dieser Abschnitt „Die Architektur des MC 68000“ genannt (und nicht: „Die Architektur des MC 68008“).

Die interne 32-Bit Architektur des MC 68008 besteht aus siebzehn 32-Bit Registern, zwei 32-Bit Stack-Pointern, einem 32-Bit Programm-Zähler und einem 16-Bit Status Register. Am nächsten kommt diesem Umfang lediglich der Mikroprozessor von National Semiconductors, die NS 16000 Serie (der wohl von Acorn als einer ihrer zweiten Processoren benutzt werden wird). So hat z.B. der NS 16032 acht 32-Bit Register und eine fast gleiche Anzahl von 24-Bit Registern.

Die siebzehn 32-Bit Register des MC 68008 machen ihn zu einem außergewöhnlichen Mikroprozessor und die Bausteine aus der MC 68000 Serie sind tatsächlich die einzigen Mikroprozessoren, bei der alle wichtigen Register über 32 Bit verfügen.

Der Intel i8088 (wie er im IBM PC verwendet wird) hat 16-Bit Register, aber die hat auch der Motorola MC 6809 (der im Tandy Color Computer und im Dragon 32 arbeitet). Beide, sowohl der i8088 als auch der MC 6809, haben einen 8-Bit Daten-Bus, trotzdem spricht man vom ersteren als einem 16-Bit Chip und deklariert den letzteren als 8-Bit Chip.

Zum Verständnis der Architektur des MC 68000 müssen wir erst einmal das „Programmierungs-Modell“ untersuchen. Es wird mit MC 68008/MC 68000 bezeichnet, weil die beiden Register-Architekturen identisch sind. Neuere und noch komplexere Versionen des MC 68000 (die MC 68010 und 68020 Chips) haben etwas abweichende Architekturen. Es gibt aber keine internen Unterschiede zwischen dem MC 68008 und dem MC 68000, alle Unterschiede sind extern.

Der erste Registerblock enthält acht Datenregister (bezeichnet mit D 0 bis D 7), die für Byte- (8 Bit), Wort- (16 Bit) und Lang-Wortoperationen benutzt werden können. Der Befehlssatz des MC 68000 (s.a. Kapitel 8) enthält

Instruktionen für arithmetische Operationenn mit allen Datenlängen, wenn auch die Hardware-Befehle für die Multiplikation und Division spezielle Datentypen verlangen.

Jedes Daten-Register umfaßt 32-Bit: Byte-Werte lagern in den unteren 8 Bits (Bit 0 bis 7); Worte in den unteren 16 Bit (also Bit 0 bis 15); und Lang-Worte belegen das ganze Register (Bit 0 bis 31). Bit 0 liegt rechts außen und repräsentiert den niedrigsten Wert, und das Bit mit dem höchsten Wert liegt links außen und ist als Bit 31 gekennzeichnet.

Wird z.B. ein Daten-Register für eine Operation benutzt, die Byte-Werte erwartet, dann werden nur die Bits 0 bis 7 modifiziert; die anderen Bits des Daten-Registers (also Bit 8 bis 31) sind durch diese Operation nicht betroffen und werden nicht verändert. Dieses ist einer der Punkte, in denen sich Daten-Register von Adress-Registern unterscheiden.

Für Adressierungen, die die Benutzung „indirekter Adress-Register mit Index“ erfordern, können Daten-Register verwendet werden (s.a. Kapitel 8), und auch Adress-Register können als Index-Register dienen. Es gibt acht Adress-Register (A 0 bis A 7), wovon sieben (A 0 bis A 6) allgemeine Adress-Register sind und eines (A 7) gleichzeitig zwei Stack-Pointer enthält. Das Adress-Register A 7 wird von vielen Befehlen indirekt benutzt, die den Stack-Pointer benötigen. Wird das Register A 7 direkt angesprochen, bewirkt dies wiederum eine Änderung des System-Stack-Pointers. Ob der Benutzer-Stack-Pointer (USP) oder der Supervisor-Stack-Pointer (SSP) verändert wird, ist vom Privileg-Status des Prozessors abhängig (wie schon erläutert wurde: vom Zusammenspiel der FC Codes und dem S-Bit des Status-Registers).

Im Gegensatz zu den flexiblen Daten-Registern kennen die Adress-Register nur 32-Bit Werte (nur Lang-Worte). Gelegentlich erfordern Instruktionen nur Wort-Operationen, dann wird automatisch der richtige Teil des Registers benutzt, aber erst dann, wenn der Rest des Registers zur Sicherung des Vorzeichens modifiziert ist (das wird im Kapitel 8 ausführlicher erklärt).

Bei der Verwendung der Postincrement- und der Predecrement-Adressierung (Adressierung mit nachfolgender oder vorheriger Erhöhung des Index-Registers, d. Übersetzer) ist es möglich, Adress-Register als Stack-Pointer zu benutzen (s.a. Kapitel 8), oder Adress-Register als Basis-Adress-Register anzusprechen (eine Basis-Adresse ist eine bestimmte Stelle im Speicher, z.B. der Anfang einer Tabelle).

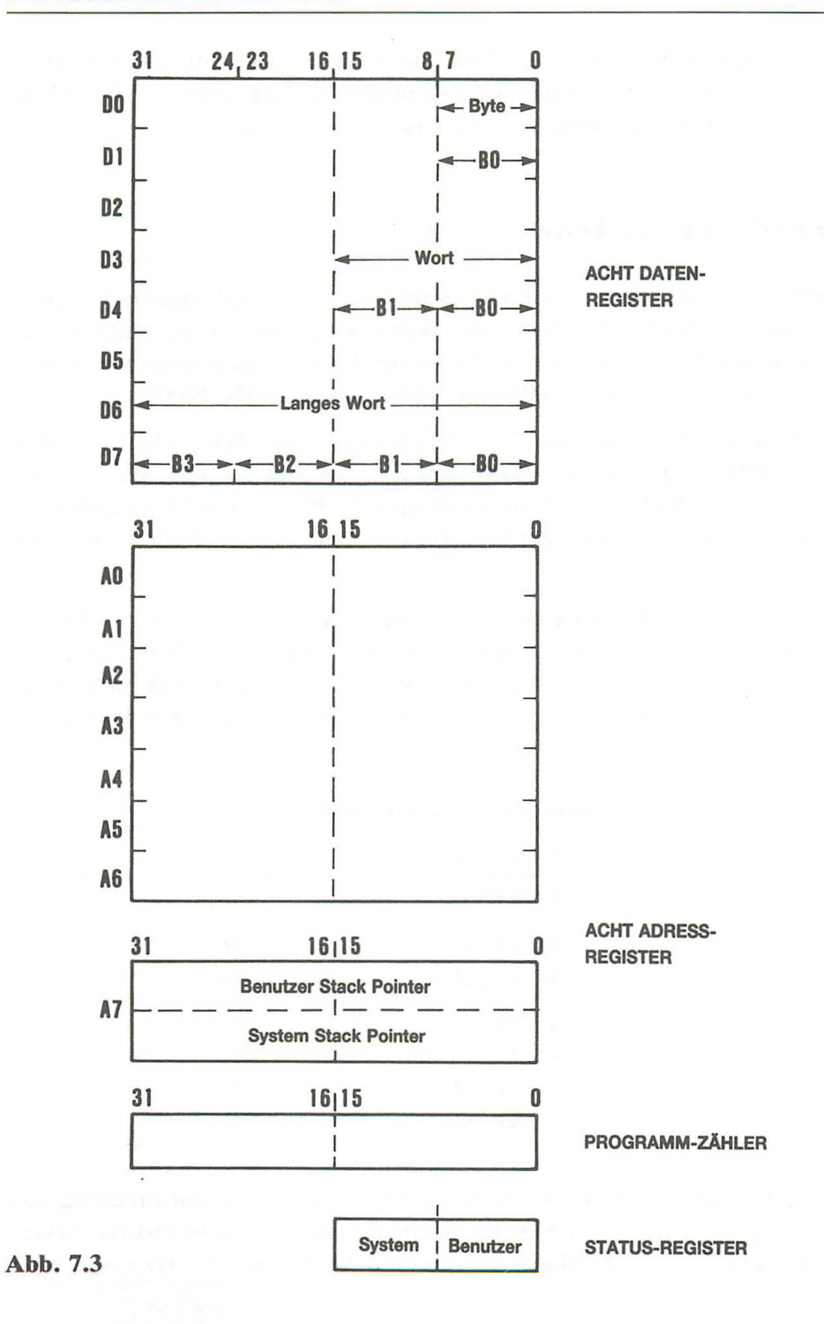

31
24
23
16
15
8
7
0
D0
D1
D2
D3
D4
D5
D6
D7
Byte
B0
Wort
B1
B0
Langes Wort
B3
B2
B1
B0
ACHT DATEN-
REGISTER
A0
A1
A2
A3
A4
A5
A6
ACHT ADRESS-
REGISTER
A7
Benutzer Stack Pointer
System Stack Pointer
PROGRAMM-ZÄHLER
System
Benutzer
STATUS-REGISTER

Abb. 7.3

Obwohl die Adress-Register 32 Bits umfassen, berücksichtigt der MC 68008 nur zwanzig Bits. Wenn eine Adresse zu groß ist, dann wird sie „abgeschnitten“, indem nur die unteren 20 Bits akzeptiert werden.

Zugriff auf die Daten

Jedes Byte im Speicher des MC 68008 kann individuell adressiert werden, wobei das höherwertige Byte eines Wortes immer eine gerade Adresse hat, die auch immer die Adresse des Wortes ist. Es ist übrigens leichter zu verstehen, wie das beim MC 68008 funktioniert, als beim MC 68000.

Die Adressierung von Speicherstellen wurde bereits besprochen, trotzdem zur Erinnerung: beim MC 68008 gibt es einen Stift für jedes Bit von 0 bis 19, wogegen der MC 68000 einen Stift für jedes Bit von 0 bis 23 hat, außerdem noch weitere zwei Stifte zur Steuerung, ob ein Byte oder ein Wort geholt werden soll.

In Abbildung 7.3 können wir sehen, wie ein Lang-Wort in einem Register gespeichert wird (von links nach rechts), und zwar in der Reihenfolge B 3, B 2, B 1, B 0, aber das entspricht nicht der Art, wie die Bytes im Speicher deponiert werden. In Tabelle 7.4 ist der Wert von ADRESSE immer eine gerade Zahl.

Tabelle 7.4 Datenspeicherung beim MC 68008

Typ	Speicherstelle	Inhalt
Byte	ADRESSE	B0
Wort	ADRESSE	B1
	ADRESSE + 1	B0
Lang-Wort	ADRESSE	B3
	ADRESSE + 1	B2
	ADRESSE + 2	B1
	ADRESSE + 3	B0

Instruktionen und Mehrbyte-Daten werden in Gruppen, entsprechend den Datentypen, Byte für Byte übernommen. So wird z.B. im Wort-Lese-Zyklus das Byte an der geradzahligen Adresse (also ADRESSE bzw. Byte B1) zuerst

gelesen und anschließend das Byte mit der nächstfolgenden ungeraden Adresse.

Zur Übernahme eines Wortes benötigt der MC 68000 vier Zyklen, wogegen der MC 68008 des QL acht Zyklen braucht. Abbildung 7.4 zeigt das Fluß-Diagramm zum Lesen eines Wortes beim MC 68008, und zum Vergleich sehen Sie in Abbildung 7.5 das Äquivalent für den MC 68000.

Betrachtet man die meisten Vergleichs-Zeiten, dann ist der Daten-Zugriff und die Datenmanipulation beim MC 68000 ungefähr zweimal so schnell wie beim MC 68008. Bei den Zeiten für viele andere Instruktionen ist der MC 68008 ungefähr halb so schnell wie der MC 68000, obwohl der Unterschied bei Byte-Instruktionen oft geringer ist.

Wenn der MC 68008 Prozessor auch langsamer als der MC 68000 ist, der Chip im QL ist trotzdem schneller als der Chip im IBM PC (dem i8088).

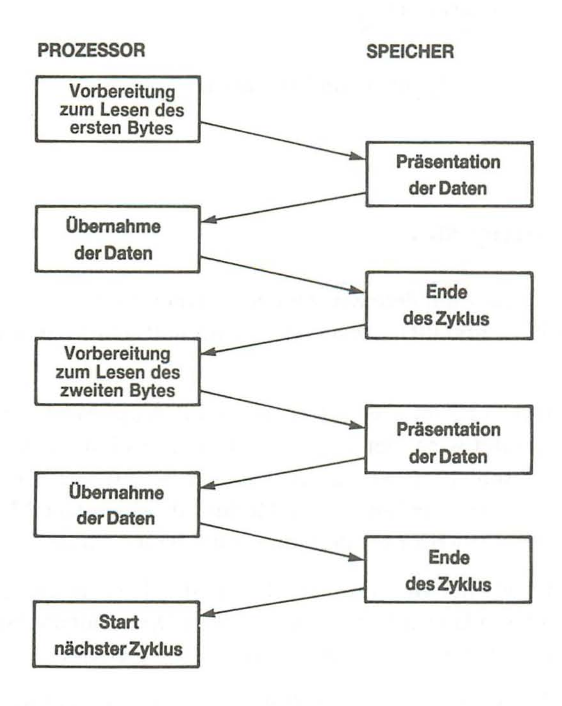


Abb. 7.4 Fluß-Diagramm für das Lesen eines Wortes beim MC 68008

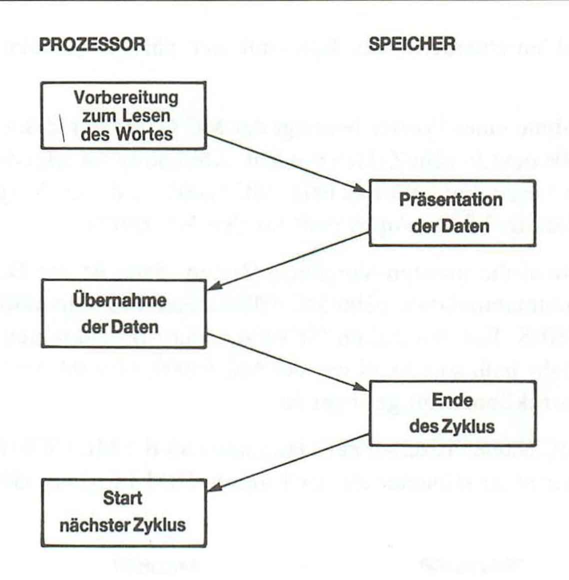


Abb. 7.5 Wort-Lese-Zyklus beim MC 68000

Das Status-Register

In Übereinstimmung mit dem Konzept des Privileg-Status ist das Status Register (das ein Wort umfaßt) in zwei Bytes unterteilt: ein System-Byte und ein Benutzer-Byte.

Das System-Byte erlaubt das Setzen einer Interrupt-Maske, so daß jeder Interrupt bearbeitet wird, der einen höheren Status hat als der Wert in der Maske (Level 7 Interrupts werden aber immer bearbeitet). Ein Flag-Bit (S) gibt an, ob das System im Supervisor-Modus oder Benutzer-Modus arbeitet und ein weiteres Flag-Bit (T) zeigt an, ob der Trace-Mode aktiviert ist.

Das Benutzer-Byte hat vier normale Flags und außerdem das „Extend" (X) Flag. Dieses letzte Flag wird bei arithmetischen Operationen mit mehrfacher Präzision benutzt (und spiegelt das „Carry" Flag wieder).

So gibt es z.B. die Instruktion „ANDI to SR" (das ist ein logisches UND zum Status-Register), eine der Instruktionen, die der Programmierer nicht

in Benutzer-Programmen verwenden kann. Es ist aber möglich, einen „MOVE SR to EA" vorzunehmen (einen Transfer vom Status-Register zur effektiven Adresse) und so die Analyse des Status-Registers zu ermöglichen.

Abbildung 7.6 zeigt detailliert den Aufbau des Status-Registers.

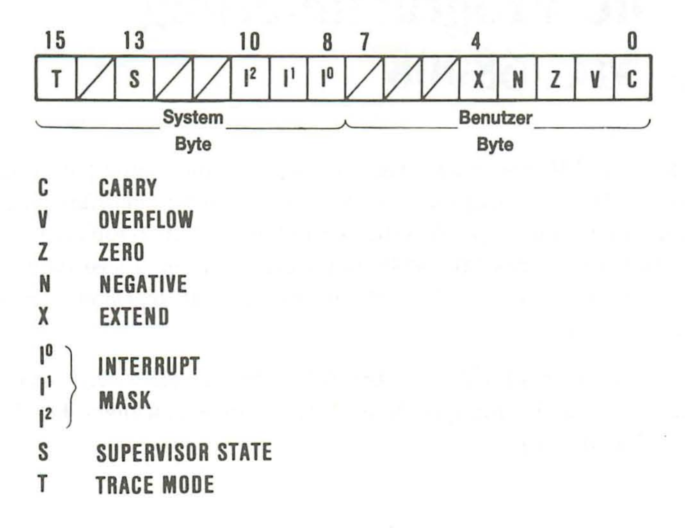


Abb. 7.6 Das Status-Register

8. Die Programmierung des MC 68008

Der Motorola MC 68008 hat, wie wir gesehen haben, eine gut durchdachte und recht gefällige Mikroprozessor-Architektur. Aber ein Mikroprozessor ist nicht nur Architektur – er soll Arbeit verrichten – deshalb soll in diesem Kapitel der Befehlssatz des MC 68000 untersucht werden. Es ist dabei wichtig, immer daran zu denken, daß der Befehlssatz des MC 68008 mit dem des MC 6800 identisch ist.

Um das Verständnis für die MC 68000-Befehle zu erleichtern, werden Vergleiche mit den drei bereits genannten 8-Bit Prozessoren angeführt (MC 6809, MCS 6502 und Z80).

Das Swap-Problem (Austausch)

Bevor wir die Befehle genauer ansehen, wollen wir mit einer einfachen Anwendung beginnen, und zwar mit dem Bubble-Sort, der uns schon im Kapitel 2 beschäftigt hat. Zunächst wollen wir nur einen bestimmten Teil des Bubble-Sorts studieren, den Austausch von Elementen. Wenn wir unterstellen, daß die Elemente EINS und ZWEI heißen, dann können wir sie unter Benutzung eines temporären Zwischenspeichers TEMP austauschen:

TEMP ← EINS
EINS ← ZWEI
ZWEI ← TEMP

(Der Pfeil ← zeigt an, daß der Wert des rechten Elements im linken Element gespeichert werden soll).

Wenn wir unterstellen, daß die auszutauschenden Werte als einzelne Bytes gespeichert sind, gelten beim MC 68000 folgende Befehle:

```
MOVE.B  (A6), D6
MOVE.B  D6, -1 (A6)
MOVE.B  D7, (A6)
```

Einzeln betrachtet erklären sich die Befehle fast selbst. Das Adress-Register A6 (geschrieben: (A6)) enthält die Adresse des Bytes, das dem Element EINS entspricht. Man sagt auch, A6 „zeigt" auf die entsprechende Speicherstelle. Das Daten-Register D7 enthält den Wert des Bytes, der dem Element ZWEI entspricht, und D6 wird als Äquivalent für TEMP benutzt.

Der Befehl:

```
MOVE.B  (A6), D6
```

transferiert den Inhalt des Bytes, auf das A6 zeigt, zum Daten-Register D6. Das bedeutet, daß wir den Wert jetzt zweimal haben, einmal an der Speicherstelle, auf die Register A6 zeigt und einmal im Register D6. Benutzer des Z80 werden bemerken, daß die Bedeutung der Adressen hier umgekehrt ist.

Der nächste Befehl

```
MOVE.B  D6, -1(A6)
```

bringt den Wert im Register D6 an die Speicherstelle, deren Adresse um 1 geringer ist als die, auf die A6 zeigt. Der Inhalt von A6 wird dabei nicht verändert. Der Wert, der ursprünglich an der Stelle gespeichert war, deren Adresse in A6 steht, ist jetzt auch in der Speicherstelle davor zu finden.

Wenn der Inhalt von A6 $005067 ist (das Vorzeichen $ gibt an, daß die folgende Zahl eine hexadezimale Zahl ist), dann ist der Wert in dem Byte an der Adresse $005067 gemeint. Der erste Befehl lädt das Daten-Register D6 mit dem Inhalt der Speicherstelle $005067 und der zweite Befehl lädt das Byte an der Speicherstelle $005066 (= $005067 – $1) mit dem Inhalt des Registers D6.

Jetzt ist der gleiche Wert in zwei nebeneinander liegenden Stellen gespeichert, wobei A6 auf die größere Adresse zeigt. Der verbleibende Wert (im Register

D7) muß an dieser größeren Adresse deponiert werden (ursprünglich war der Wert an der niedrigeren Adresse gespeichert). Der Befehl

```
MOVE.B  D7, (A6)
```

schreibt jetzt den Inhalt des Registers D7 in die Speicherstelle mit der Adresse, die in A6 steht.

Hinzufügung eines Vergleichs:

Natürlich kann man die Befehlsfolge so erweitern, daß sie einen Vergleich der in den Bytes gespeicherten Werte einschließt:

```
MOVE.B  (A6) +, D7
CMP.B   (A6), D7
BCC.S   LABEL
```

Diese Befehle werden den Befehlen zum Datentausch vorgeschaltet und bereiten das System entsprechend vor.

Der erste Befehl:

```
MOVE.B  (A6) +, D7
```

lädt den Byte-Wert, der an der Stelle gespeichert ist, auf die das Register A6 zeigt (z.B. $005066), in das Daten-Register D7 und erhöht den Inhalt des Adress-Registers A6 automatisch um den Wert Eins (z.B. auf 005067). Diese automatische Erhöhung des Wertes in A6 erfolgt, weil MOVE.B ein Befehl zur Manipulation von Daten-Bytes ist. Das führt dazu, daß A6 jetzt auf die nächsthöhere Speicherstelle zeigt.

Da der Wert in A6 um „Eins" erhöht wurde, vergleicht der Vergleichsbefehl:

```
CMP.B (A6), D7
```

den Inhalt des Registers D7 mit dem Inhalt der Speicherstelle, die eine Stelle weiter liegt als die ursprünglich von A6 angezeigte. Der Vergleich findet zwischen Bytes statt, da der Vergleichsbefehl CMP den Zusatz .B hat.

Der letzte der drei Befehle:

```
BCC.S.  LABEL
```

prüft das Ergebnis des gerade erfolgten Vergleichs und verzweigt an eine Stelle im Programm, die durch die symbolische Adresse LABEL gekennzeichnet ist. Voraussetzung für die Verzweigung ist jedoch, daß das Carry-Flag „Null" ist. Diese Bedingung ist erfüllt, wenn der Inhalt von D7 größer als der Inhalt der Speicherstelle ist, auf die A6 zeigt. Der Anhang .S bedeutet, daß es sich um einen kurzen Sprung (= short jump) handelt (im Bereich von − 128 bis + 127 Adressen).

Insgesamt sieht unser Programm jetzt so aus:

```
MOVE.B  (A6) +, D7
CMP.B   (A6), D7
BCC.S   LABEL
MOVE.B  (A6), D6
MOVE.B  (D6), - 1 (A6)
MOVE.B  D7, (A6)
```

und damit sollte klar sein, daß der Befehl MOVE.B sehr flexibel ist.

Noch nicht so klar ist vielleicht die Tatsache, daß eine Änderung der obigen Befehlsfolge zur Bearbeitung von Lang-Worten (gleichzeitig 4 Bytes) sehr einfach ist:

```
MOVE.L  (A6) +, D7
CMP.L   (A6), D7
BCC.S   LABEL
MOVE.L  (A6), D6
MOVE.L  (D6), - 4 (A6)
MOVE.L  D7, (A6)
```

Die Änderung ist so geringfügig, daß man die zweite Version versteht, wenn man die erste verstanden hat: der Hauptunterschied (außer der Kennzeichnung für Lang-Worte .L) besteht darin, daß die Adresskorrektur jetzt − 4 statt − 1 beträgt.

Einige äquivalente Routinen für 8-Bit Prozessoren:

Nachstehend ein Beispiel, wie man das Gleiche für den Z80 kodieren könnte:

```
LD   A, (IX)
LD   D, A
LD   E, (IX + 1)
CP   E
JR   NC, LABEL
LD   (IX), E
LD   (IX+1)
INC  IX
```

Der Inhalt der Speicherstelle, deren Adresse im Register IX steht, wird in den Akkumulator A geladen und dann in das Register D transferiert. Anschließend wird das Register E mit dem Inhalt der folgenden Speicherstelle geladen. Register E wird dann mit dem Inhalt des Akkumulators verglichen. Ist danach das Carry-Flag nicht gesetzt, erfolgt die Verzweigung zu LABEL. Wenn das Carry-Flag gesetzt ist, dann wird der Inhalt von E an der Stelle gespeichert, auf die IX zeigt und der Inhalt von D wird in der nächsten Speicherstelle abgelegt. Zum Abschluß wird dann IX um „Eins" erhöht. Diese Logik ist etwas anstrengender als beim MC 68000 und nicht so elegant.

Auch der MCS 6502 ist ein populärer 8-Bit Mikroprozessor, für den die Befehle so geschrieben werden müßten:

```
LDA  (START), Y
INY
CMP  (START), Y
BCC  LABEL
TAX
LDA  (START), Y
DEY
STA  (START), Y
TXA
INY
STA  (START), Y
```

Diese Befehlsfolge scheint noch komplizierter zu sein. Bei Betrachtung dieser Sequenz sollte man aber daran denken, daß der MCS 6502 nur einen Akku-

mulator und zwei Indexregister (X und Y) hat, so daß diese Routine in Wirklichkeit also nicht so kompliziert ist.
Es findet ein Austausch zwischen Speicherstellen, deren Anfangsadresse an der Speicherstelle START steht, dem Akkumulator und dem X-Register statt. Das Y-Register arbeitet als Zähler, der angibt, wie groß die Entfernung zwischen der Anfangsadresse und dem augenblicklich relevanten Byte ist.
Solange das Programm, von dem diese Sequenz ein Bestandteil ist, nicht modifiziert wird, kann die Anzahl der zu sortierenden Werte 255 Elemente nicht überschreiten.

Die letzte 8-Bit Routine gilt für den MC 6809, den älteren, kleineren Bruder des MC 68008:

```
LDA   , X
CMPA  , + X
BLT   LABEL
LDB   , X
STB   -1, X
STA   , X
```

und das ist die 8-Bit Routine, die auch für Anfänger noch am übersichtlichsten und am leichtesten zu verstehen ist.
Lade Akkumulator A mit dem Wert in der Speicherstelle, auf die das Register X zeigt und vergleiche den Wert mit dem Inhalt der nächsten Speicherstelle.
Stehen die Werte in der richtigen Reihenfolge, dann verzweige nach LABEL, andernfalls lade den Akkumulator B mit dem Inhalt der Speicherstelle, auf die das Register X jetzt zeigt (eine Stelle hinter LDA).
Speichere den Inhalt des Akkumulators B an die Adresse, die sich aus dem Inhalt des X-Registers minus 1 ergibt und speichere den Inhalt des Akkumulators A an der höheren Adresse. Der Befehlssatz des MC 6809 ist eindeutig der beste unter den 8-Bit Maschinen und am leichtesten zu benutzen.

Nachdem wir uns die Byte-Schaufelei bei verschiedenen 8-Bit Maschinen betrachtet haben, können wir uns vorstellen, wie komplex ein Programm zur Bearbeitung langer Worte bei ihnen sein müßte; ein Programm, das für den MC 68008 unter Benutzung des MC 68000 Befehlssatzes so einfach zu schreiben ist.

Die implizite Adressierung

Das Speichern und Zurückholen von Werten kann beim MC 68008 auf viele verschiedene Arten gesteuert werden.

Die zwei einfachsten Adressierungsmethoden (also die Angabe der Adresse einer bestimmten Speicherstelle) sind die „inherente“ (= unbestimmte) und die „direkte“ Register-Adressierung.

Eine unbestimmte Adressierung liegt dann vor, wenn keine Adresse angegeben werden muß: das System weiß, was zu tun ist und wo es zu geschehen hat. Der offensichtlichste Fall einer unbestimmten Adressierung ist der Befehl RTS (Rücksprung aus Subroutine). Der Rücksprung erfolgt zu dem Befehl, der auf den JSR folgt (Sprung zur Subroutine), der das Unterprogramm aktiviert hatte.

Bei der RTS-Anweisung wird die Rücksprung-Adresse aus dem Adress-Stack übernommen, sie wird deshalb im Befehl nicht benötigt. Bei allen Fällen mit unbestimmter Adressierung (die Traps und Returns von den „Ausnahmen“ mit einschließen) verwendet der Prozessor entweder den Benutzer-Stack-Pointer, um Daten vom und zum Speicher zu bewegen, oder den System-Stack-Pointer, um den Inhalt des Programm-Zählers zu speichern.

Viele Befehle des MC 68000 spezifizieren die internen Register des Prozessors und benötigen deshalb keine Adress-Angaben. Bei der direkten Adressierung über Register gibt es zwei Möglichkeiten: direkt über Daten-Register, oder direkt über Adress-Register, wobei bestimmte Befehle nur eine spezielle Mischung von Daten- und Adress-Registern zulassen.

Aus Abbildung 8.1 ist zu ersehen, wie die „unbestimmte“ und die „Register direkte“ Adressierung funktionieren. Ein Viereck mit Diagonalen repräsen-

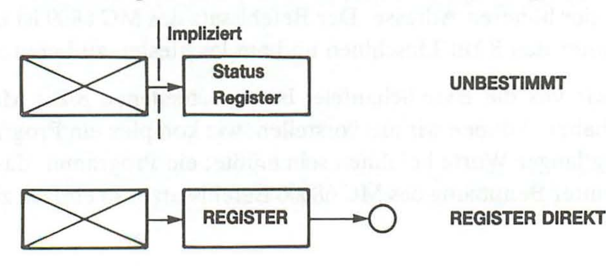


Abb. 8.1 Die impliziten Adressierungen

tiert einen Befehl, ein Viereck ohne Diagonalen stellt entweder ein Register (SP, PC, A oder D) oder eine Speicherstelle (M) dar, und ein Kreis steht für einen Wert, nicht für eine Adresse.

Zur Wiederholung: bei der unbestimmten Adressierung zeigt der Befehl entweder auf das Benutzer-Stack-Pointer Register (USP), welches wiederum auf eine Adresse zeigt, oder auf das System-Stack-Pointer Register (SSP), das ebenfalls auf eine Adresse zeigt.

Im ersten Beispiel ist die Adresse, auf die der USP zeigt, im Programm-Zähler-Register geladen, und an dieser Adresse wird der Programmablauf fortgesetzt. Im zweiten Fall zeigt der SSP auf eine Adresse und an dieser Adresse ist der Inhalt des Programmzählers gespeichert, um nach Aufarbeitung einer „Ausnahme“ (z.B. Interrupt) zur Verfügung zu stehen.

Absolute Adressierungs-Arten

Die nächste Form der Adressierung ist (genau genommen) keine wirkliche Adressierung, da keine Adresse angegeben wird. Man nennt sie die unmittelbare Adressierung (= immidiate adressing).

Bei der unmittelbaren Adressierung zeigt der Befehl unmittelbar auf einen Wert, nicht auf eine Adresse. Betrachten Sie die folgende Instruktion:

```
ADD.W  #$090A, D5
```

die eine Kombination von Adressierungen enthält. Dieser Befehl addiert den Wert $090A zum Inhalt des Registers D5, und die Additions-Operation erfolgt auf der Wort-Ebene. Wort-Operationen werden grundsätzlich unterstellt (= Default), wenn keine andere Spezifikation angegeben wird, deshalb kann der Befehl auch so geschrieben werden:

```
ADD  #$090A, D5
```

Die Empfangsadresse ist ein Register, daher wird diese Art der Adressierung auch Register-Adressierung genannt. Für einen Typ des ADD-Befehls ist es die einzige zulässige Angabe der Empfangsadresse.

Die Sendeadresse ist keine Adresse, sondern ein Wert und ist ein typisches Beispiel der unmittelbaren Adressierung. Die Tatsache, daß #$090A keine Adresse, sondern ein Wert ist, ergibt sich aus dem Kennzeichen #. Der Grund, warum das die unmittelbare Adressierung genannt wird, ist darin zu finden, daß der Wert im Speicher unmittelbar auf den Befehl folgt.

Es ist natürlich zu beachten, daß eine Empfangsadresse niemals eine unmittelbare Adresse sein kann.

Die Adresse des Wertes ist durch den Befehl bekannt, denn der Wert steht im Speicher direkt hinter dem Befehl; sie wird also durch die Position relativ zum Befehl erkannt.

Die nächste Adressierungs-Art geht noch etwas weiter mit der Idee der „Adresse relativ zum Befehl".
Diese Methode heißt „absolute kurze Adressierung" (= absolute short) und ist der Page-0-Adressierung einiger 8-Bit-Mikroprozessoren sehr ähnlich. Mit der „absolute short" Adressierung kann man die unteren 32 K des Speichers (Adressen $ 00000 bis $ 07FFF) und die obersten 32 K (Adressen $ F8000 bis $ FFFFF) adressieren.
Der Grund für diese weite Spanne zwischen den Speicherstellen ist vielleicht am besten zu verstehen, wenn man bedenkt, daß diese beiden Teile des Speichers (so wie der MC 68008 arbeitet) als zusammenhängend angesehen werden. Die Speicherstelle $ FFFFF wird als „Nachbar" der Speicherstellen $ FFFFE (eine niedriger) und $ 00000 (eine höher) betrachtet.

Bei der „absolute short" Adressierung ist die zum Befehl gehörende Adresse ein Wort groß (also 2 Bytes oder 16 Bits), im Adress-Register hat sie jedoch zweiunddreißig Bits. von diesen zweiunddreißig Bits sind jedoch nur die ersten zwanzig für den MC 68008 relevant (hier unterscheidet er sich vom MC 68000, bei dem vierundzwanzig Bits relevant sind).

Wenn das höchstwertige Bit der kurzen sechzehn Bit Adresse 0 ist (also eine Adresse von $ 0000 bis $ 7FFF angibt), dann wird das Adress-Register mit Nullen in den oberen Bits aufgefüllt (Adresse $ 00000000 bis $ 00007FFF).
Wenn aber das höchstwertige Bit 1 ist, also eine Adresse zwischen $ 8000 bis $ FFFF anzeigt, dann wird der Rest des Adress-Registers auch auf Eins geschaltet, so daß Adressen von $ FFFF8000 bis $ FFFFFFFF zustande kommen. Soweit es nicht anders spezifiziert wird, ist das die Default-Form der Adresse. Beachten Sie, daß, obwohl die Adresse in einem Register z.B.

$ FFFF9999 ist, diese Adresse im MC 68008 in die Adresse $ F9999 konvertiert wird.
Zum Beispiel wird der Befehl

```
ADD.W  $090A, D5
```

den an den Wort-Speicherstellen mit der Adresse $0090A gespeicherten Wert zum Inhalt des Registers D5 addieren. Würde der Befehl

```
ADD.W  $890A, D5
```

lauten, dann würde der an den Wort-Speicherstellen mit der Adresse $F890A gespeicherte Wert zum Inhalt des Registers D5 addiert.

Will man den Inhalt der Wort-Speicherstellen mit der wirklichen Adresse $890A addieren, dann darf nicht die Kurz-Adresse angegeben werden, sondern es muß die volle Adresse geschrieben werden:

```
ADD.W $0890A, D5
```

Dieser Befehl addiert den Inhalt der Wortspeicherstellen mit der Adresse $0890A zum Inhalt des Registers D5.

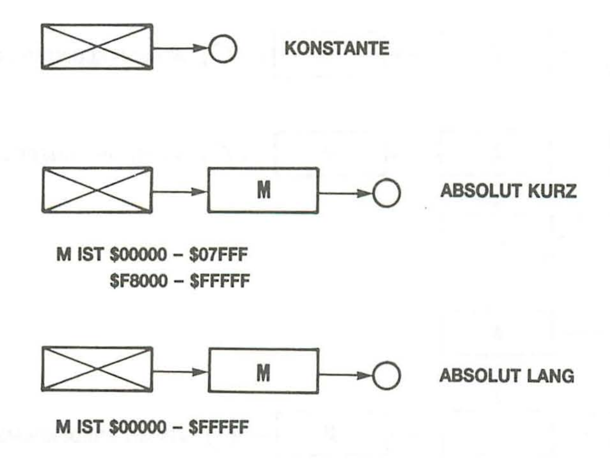


Abb. 8.2

Diese neue Form der Adressierung im letzten Befehl nennt man absolute „lange“ Adressierung, denn der ganze Umfang des Speichers kann damit adressiert werden.

Zur Durchführungszeit, die die Auswertung der Adressen beansprucht: die Differenz für Wort-Speicherstellen ist 16 Einheiten (Perioden) für Kurzadressen und 24 Einheiten (Perioden) für lange Adressen.

Abbildung 8.2 zeigt die beiden Adressierungs-Arten schematisch.

Die indirekte Adressierung

In der Bubble Sort Routine hatten wir den Befehl

```
MOVE.B  (A6), D6
```

der bedeutet, daß der Wert in dem Byte, dessen Adresse im Register A6 steht, in das Daten-Register D6 geladen wird. Diese Angabe der Sende-Adresse nennt man „indirekte“ (über Adress-Register) Adressierung.

In Abbildung 8.3 ist diese Methode als ein Befehl dargestellt, der auf ein Adress-Register zeigt, das wiederum zeigt auf eine Speicherstelle, und diese

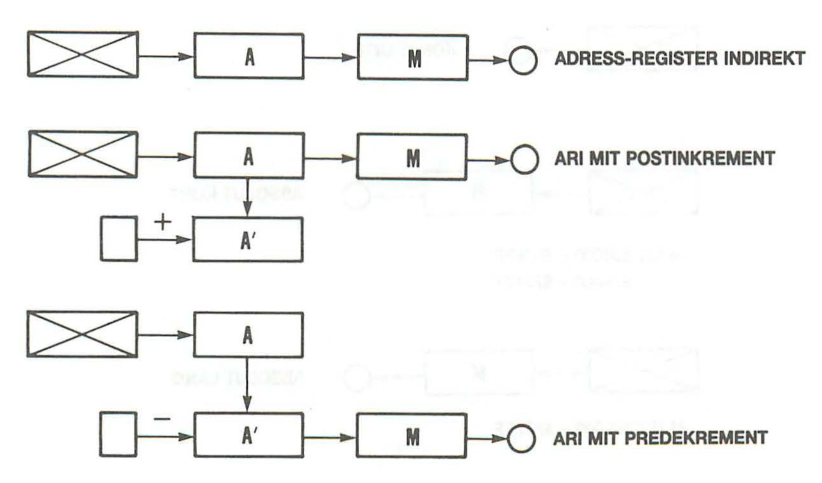


Abb. 8.3 Methoden der indirekten Adressierung

Speicherstelle enthält einen Wert. Abhängig vom Befehl kann die indirekte Adressierung entweder für die Sende- oder für die Empfangs-Adresse verwendet werden, zum Beispiel:

```
MOVE.B  D7,(A6)
```

hier wird der Wert im Register D7 in das Byte transferiert, auf das das Adress-Register A6 zeigt.

```
MOVE.B  (A6)+,D7
```

ist ein Beispiel für die indirekte Adressierung mit nachfolgender Erhöhung des Inhaltes des Adress-Registers.
Diese Art entspricht an sich der „normalen“ indirekten Adressierung. Der Unterschied besteht darin, daß der Inhalt des Adress-Registers automatisch um den Wert erhöht wird, der dem Datentyp entspricht (in diesem Fall ist es der Wert „Eins“, denn der Datentyp ist Byte bzw. eine Speicherstelle).

Das ist das gleiche, als hätten wir zwei Befehle:

```
MOVE.B  (A6), D7
ADDA    #$1, A6
```

also, bewege den Wert aus einer Speicherstelle, indirekt adressiert durch das Adress-Register, in ein Daten-Register (der erste Befehl) und dann addiere einen direkten Wert (in dem Fall $1 oder 1) zum Adress-Register A6.

Ganz allgemein gesehen bedeutet es erheblich weniger Arbeit, wenn das Programm an einen anderen Datentyp (z.B.: Worte oder Lang-Worte) angepaßt werden soll. Die verschiedenen Längen der Datentypen werden bei der automatischen Erhöhung des Wertes im Adress-Register vom Prozessor berücksichtigt.
So müßten z.B. die beiden Befehle für den Datentyp Lang-Wort (ein langes Wort erstreckt sich über vier Speicherstellen) folgendermaßen umgestellt werden, um die automatische Erhöhung des Adress-Registers zu berücksichtigen:

```
MOVE.L  (A6), D7
ADDA    #$4, A6
```

Bei Verwendung der Möglichkeit der automatischen Erhöhung im Adress-Register genügt eine einzige Instruktion, die fast der ursprünglichen MOVE.B Anweisung entspricht, da sich nur die Befehlserweiterung ändert:

```
MOVE.L  (A6)+, D7
```

Die nachfolgende automatische Adress-Erhöhung kann, abhängig vom Befehl, sowohl für den Sender als auch für den Empfänger benutzt werden. Die automatische Erhöhung um einen festen Wert ist in Abbildung 8.3 durch ein Quadrat dargestellt.

Im Bubble Sort hatten wir einen Befehl, der auch als

```
MOVE.B  D6,-(A6)
```

geschrieben werden könnte, das entspricht den Befehlen

```
SUBA    #$1, A6
MOVE.B  D6, (A6)
```

Hierdurch wird der Inhalt des Daten-Registers D6 in das Byte des Speichers geladen, auf das das Adress-Register A6 zeigt, nachdem zuvor der Inhalt des Adress-Registers um den Wert $1 reduziert wurde.
Es gibt dafür viele Namen, aber offiziell (bei Motorola) heißt es „indirekte Adressierung mit vorheriger Verringerung“. Diese Adressierungs-Art ist mit der zuvor besprochenen indirekten Adressierung mit nachfolgender Erhöhung sehr eng verwandt. Der Ablauf der Adressierung ist aus Abbildung 8.3 zu entnehmen.

Der MOVE.B mit nachfolgender Erhöhung benötigt 12 Perioden, wenn die erhöhte Adresse dem Sender gilt und der Empfänger ein Register ist. Die gleiche Operation dauert 14 Perioden bei vorhergehender Verringerung des Register-Inhalts. Wird die Adressierung für Sender und Empfänger ausgetauscht, reduziert sich die Zeit auf 12 Perioden (wie im oben angeführten Fall).
Der indirekt adressierte Befehl MOVE.B zum Daten-Register braucht auch 12 Perioden, dauert also genau so lange wie der gleiche Befehl mit automatischer Erhöhung. Wenn man die zusätzliche Zeit für den ADDA- oder SUBA-Befehl berücksichtigt, lassen sich erhebliche Einsparungen sowohl bei

der Ablaufzeit als auch bei der Anzahl an Befehlen durch Verwendung der sehr nützlichen automatischen Befehle erreichen.

Indirekte Adressierung mit Displacement und Index

Eng verwandt mit den automatisch erhöhenden oder verringernden Befehlen sind die Befehle, die es dem Programmierer erlauben, die Größe der Veränderung im Adress-Register selbst zu bestimmen.

Das einfachste Beispiel dafür haben wir in dem Befehl

```
MOVE.B  D6, -1(A6)
```

gefunden, der, wie wir uns erinnern, auch

```
MOVE.B  D6, -(A6)
```

geschrieben werden könnte und wäre dann die „indirekte Adressierung mit Verringerung". Der Original-Befehl ist ein Beispiel für indirekte Adressierung mit Displacement (Versatz) und ist schematisch in Abbildung 8.4 dargestellt.

Wenn man die indirekte Adressierung mit Displacement aus Abb. 8.4 mit der indirekten Adressierung mit Adress-Verringerung in Abb. 8.3 vergleicht, kann man einige Unterschiede feststellen. Die beiden wichtigsten sind die Form des Symbols, das auf den neuen Wert im Adress-Register zeigt, und die Benutzung eines Pseudo-Registers TEMP.
Bei der automatischen Verringerung ist das Symbol ein Quadrat, das andeutet, daß es sich um einen festen Wert handelt. Beim Displacement ist die zusätzliche Angabe dagegen ein Kreis, wodurch eine Variable angezeigt wird. Bei den beiden automatischen Veränderungen ergeben sich ständige Änderungen im Inhalt des Adress-Registers.

Der Befehl

```
MOVE.B  D6, -1(A6)
```

entspricht dem Befehl mit automatischer Verringerung um „Eins", aber für den Befehl

```
MOVE.B  $20(A6),D6
```

gibt es keine Paralelle. In diesem Beispiel wird der Hexadezimal-Wert $20 zum Inhalt des Adress-Registers A6 addiert und der Inhalt der Speicherstelle, auf die das Ergebnis zeigt, wird in das Datenregister D6 kopiert.
Das Displacement (im Beispiel $20) muß ein Wort, also sechzehn Bits, lang sein. Wenn der Wert negativ ist, wird es „Vorzeichen erweitert" und umfaßt dann ein Lang-Wort (da das Adress-Register zweiunddreißig Bits hat).

Wenn also das Displacement −1 sein soll, dann wäre das äquivalent (als Zweier-Komplement) zu FFFF. Der Betrag, der dem Adress-Register hinzugerechnet würde, wäre demzufolge FFFFFFFF, obwohl nur die ersten fünf Hex-Stellen für die Errechnung der aktuellen Adresse herangezogen werden. Da das Displacement ein Wort mit Vorzeichen ist (also 2 Bytes), sind die Limits des Displacements −32768 bis 32767 Bytes.

Der Befehl

```
MOVE.B  $20(A6,A5),D6
```

ist ein Beispiel für „indirekte Adressierung mit Displacement und Index", und eine andere Form des Befehls ist

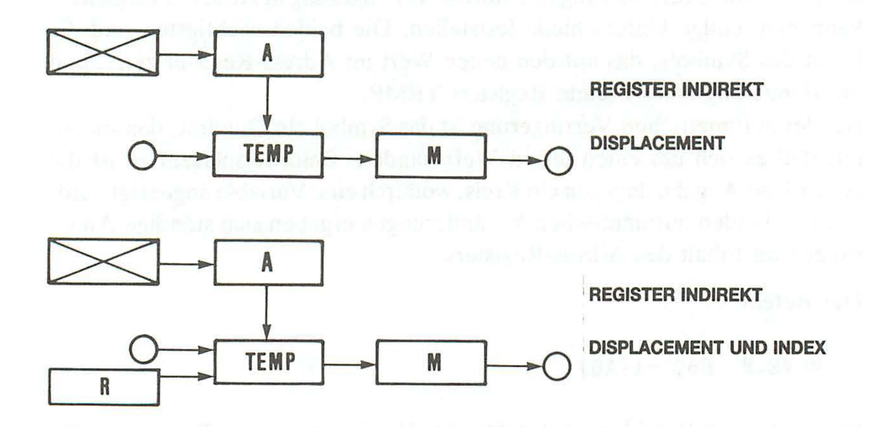


Abb. 8.4 Indirekte Adressierung mit Displacement

```
MOVE.B  $20(A6,D5),D6
```

Bei dieser Art der Adressierung ergibt sich die Adresse aus drei Elementen: dem Inhalt eines Adress-Registers, dem Inhalt eines Index-Registers (Daten- oder Adress-Register) und einem Displacement. Schematisch ist das in Abbildung 8.4 dargestellt. Allerdings besteht ein Unterschied in der zulässigen Größe des Displacements bei dieser Adressierung, da im Befehl nur ein Byte dafür zur Verfügung steht. Dieses Byte ist Bestandteil des Ein-Wort-Befehls (es ist das niederwertigere Byte), deshalb kann das Displacement (im Zweier-Komplement-Format) nur von −128 bis 127 reichen.

Der Inhalt des benutzten Index-Registers kann entweder das vollständige Lang-Wort oder nur ein Wort (2 Bytes) umfassen. Diese Adressierungsart ist sehr flexibel und erlaubt die Adressierung sehr komplizierter Datenstrukturen.

Programm-Zähler mit Displacement

Ein Beispiel hierfür ist der Befehl

```
BCC.S.  LABEL
```

und dieses ist der Zeitpunkt, an dem wir kurz die Aufgabe eines Assemblers diskutieren müssen.

Erstens, was drückt dieser Befehl aus?

Er besagt, daß eine Verzweigung zu einer Adresse, symbolisch LABEL genannt, erfolgen soll, wenn das CARRY-Flag aus (0) ist. Die Programm-Adresse, die durch die symbolische Adresse LABEL markiert wird, ist eine gewisse Anzahl Bytes von der Speicherstelle entfernt, in der der BCC.S-Code steht.
Der Befehlscode für BCC.S ist $64XX, wobei XX die Anzahl Bytes angibt, über die der Sprung auszuführen ist.
Liegt z.B. die mit LABEL markierte Programm-Adresse achtzehn Bytes (= $12) entfernt von der Adresse des Wortes, in dem die BCC.S-Instruktion steht, dann ist der komplette Befehlscode $6412. Wenn also die Adresse des

BCC.S-Befehls $2000 ist, dann befindet sich die symbolische Adresse LABEL an der absoluten Adresse $2014, also $12 plus $2 Bytes (1 Wort) vom Befehlscode.

Zweitens: woher wissen wir, daß der Befehl BCC.S LABEL als $6412 zu codieren ist?

Die Antwort darauf ist die Verwendung eines Assemblers. Ein Assembler (besonders wenn er über einen so starken Befehlssatz wie bei der MC 68000 Serie verfügt) ist eine Sprache, genau wie BASIC, FORTH oder LOGO Sprachen sind. Seine Aufgabe besteht in der Übersetzung der „Mnemonics" in den Maschinencode.

Die Benutzung eines Assemblers wurde in unserer bisherigen Diskussion einfach unterstellt, weil er das Leben viel leichter macht. Er ermittelt, daß der Befehl mit der symbolischen Adresse LABEL $14 Byte von der BCC.S LABEL-Instruktion entfernt liegt, und dann fügt er die korrekte Entfernung dem Befehl bei. Der BCC.S Befehl könnte auch

```
BCC.S  * +$12
```

geschrieben werden. Die Verwendung einer symbolischen Adresse erleichtert das Verfahren jedoch, da dem Programmierer die Berechnung abgenommen wird.

Das

*+$12

bedeutet: Nimm die Adresse aus dem Programmzähler, addiere den Wert $12 und verzweige auf die sich so ergebende Adresse, wenn die Bedingung erfüllt ist. Der Inhalt des Programmzählers entspricht zu dem Zeitpunkt der Befehls-Adresse (des BCC.S) plus 2 Byte. Der Programmzähler wird immer um 2 Byte (ein Wort) erhöht.

Der .S-Anhang bedeutet, daß es sich um eine kurze (ein Wort) Verzweigung handelt. Wenn das .S fehlt, dann unterstellt der Assembler, daß eine „lange" Instruktion gemeint ist.

Eine „lange" Instruktion ordnet dem Displacement ein ganzes Wort zu, die Grenzen sind demzufolge −32768 bis 32766 (Verzweigungen führen immer zu „geraden" Adressen). Die Grenzen einer kurzen Verzweigung liegen bei −128 bis 126 Bytes.

Die relative Adressierung über den Programmzähler mit Displacement kann auch für andere Befehle als nur für Verzweigungen verwendet werden (bei Verzweigungen wird sie unterstellt). So findet zum Beispiel der Befehl

```
MOVE.B  ENTF(PC),D7
```

den Wert der Adresse im Programmzähler, der Wert ENTF wird hinzu addiert und das Ergebnis ist die Adresse, deren Inhalt in das Daten-Register D7 kopiert wird.

In Abbildung 8.5 können Sie sehen, daß die Prozedur genau der indirekten Adressierung über Register mit Displacement entspricht, anstelle eines Adress-Registers wird jedoch der Programmzähler genommen. (Ein Assembler muß die obige Anweisung übersetzen, um daraus den entsprechenden Befehlscode zu produzieren. Verschiedene Assembler arbeiten mit geringfügig unterschiedlichen Konventionen.)

Entsprechend der indirekten Adressierung mit Register, Displacement und Index gibt es auch die indirekte Adressierung mit dem Programmzähler, Displacement und Index. Der Assembler erkennt das, wenn der Befehl im folgenden Format eingegeben wird:

```
MOVE.B $2F(PC,A6), D7
```

und die Interpretation entspricht der Adress-Register Version (s.a. Abb. 8.5).

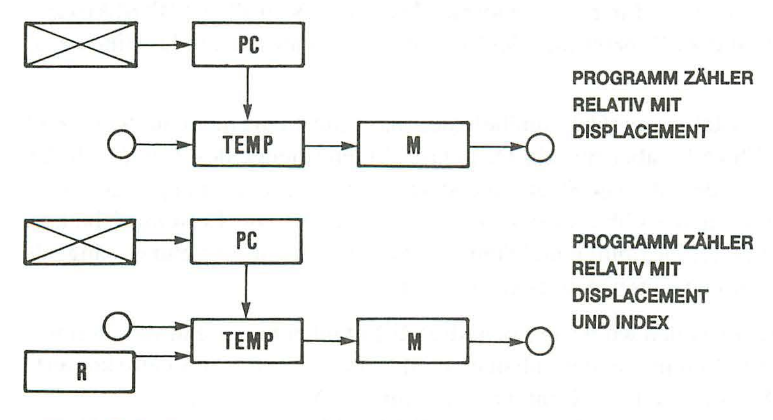


Abb. 8.5 Indirekte Adressierung über den Programmzähler

Die Fähigkeiten und Komplexität der Assembler ist sehr unterschiedlich. Wenn man bei 8-Bit-Computern auch einiges ohne Assembler direkt im Maschinencode programmieren kann, so verlangt der MC 68000 mit seinem Befehlssatz doch die Benutzung eines Assemblers. Als dieses Buch geschrieben wurde, stand noch kein Assembler für den QL zur Verfügung, der Verfasser erwartete aber eine baldige Freigabe.

Der Befehlssatz

Der Zilog Z80 Mikroprozessor (wie er im Sinclair Spectrum verwendet wird) hat ca. 156 verschiedene Befehle, der MOS Technology MCS 6502 dagegen nur 52. Es ist allerdings nicht unproblematisch, Befehle zu zählen.

Der Grund für die große Anzahl von Befehlen beim Z80 liegt darin, daß der Befehlssatz des Z80 inkonsequenter als viele Befehlssätze bei andere populären Mikroprozessoren ist. So hat z.B. der MC 68000 einen einzigen MOVE-Befehl, der in verschiedenen Formaten umfangreiche Möglichkeiten bietet. Die Unterschiede entstehen zum größten Teil durch die vielen verschiedenen Arten der Adressierung für diesen Befehl. Beim Z80 wird diese Vielfalt durch Verwendung vieler verschiedener Befehle erreicht.
Daß der MC 6809 (auch von Motorola) der stärkste 8-Bit Chip ist, wird allgemein anerkannt, obwohl dieser Chip nur 71 Basis-Maschinencode-Befehle hat. Dafür hat der MC aber eine große Anzahl von Adressierungsmöglichkeiten und erreicht dadurch seine enorme Flexibilität. Von allen 8-Bit Mikroprozessoren ist der MC 6809 auch der Chip, der am leichtesten in Maschinencode zu programmieren ist.

Der MC 68008 hat nur 56 Grundbefehle, das ist nur wenig mehr als die Anzahl beim MCS 6502, aber nur ein Drittel der Befehlsmenge des Z80. Doch der Befehlssatz des MC 68000 ist äußerst kräftig und leistungsfähig, das wurde an dem kleinen Bubble Sort Programm bewiesen. Die Leistungsfähigkeit kommt aus der Flexibilität und Simplizität der Befehle in Zusammenhang mit den gut durchdachten Adressierungsarten.

Als Beispiel wollen wir wieder den MOVE-Befehl nehmen. Für diesen simplen MOVE-Befehl, der den Inhalt einer Speicherstelle in eine andere kopiert, gibt es 80 verschiedene Kombinationen für die Adressierung.
Wenn man bedenkt, daß es drei MOVE-Versionen gibt (MOVE.B,

MOVE.W und MOVE.L), dann könnte man von 240 verschiedenen Befehlen sprechen, die alle MOVE heißen. (Genaugenommen gibt es sogar noch mehr, die die Verwendung von Bedingungs-Register, Status-Register und ähnliches einschließen. Wir wollen diese aber hier nicht berücksichtigen.)

Jeder Befehl wird als ein Wort gespeichert, als Beispiel soll das Binärformat des MOVE-Befehls in Tabelle 8.1 dienen.

Tabelle 8.1 Der Aufbau des MOVE Befehls

Bits	Funktion
15, 14	MOVE Operationscode: immer 00
13, 12	Datengröße: 01 = Byte, 11 = Wort, 10 = Lang-Wort
11-9	Empfänger Registernummer
8-6	Empfänger Adress-Code
5-3	Sender Adress-Code
2-0	Sender Registernummer

Die Bits 11-6 ergeben gemeinsam die effektive Adress-Spezifikation für den Datenempfänger und die Bits 5-0 zeigen die effektive Adress-Spezifikation für den Datensender.

In einigen Fällen ist die Nummer des Registers, das in die Bits 11-9 und/oder 2-0 eingefügt wird, wirklich eine Registernummer. In dem Fall würde z.B. das Daten-Register D6 als 110 codiert. Anhand der Adress-Code-Schlüssel unterscheidet das System dann zwischen Daten- und Adress-Register. In vielen Fällen steht dort aber keine „wirkliche" Register-Nummer.

Wenn wir die bereits diskutierten Arten der Adressierung betrachten, dann sehen wir, daß es einige illegale Kombinationen gibt. So ist z.B. das Kopieren einer Konstanten unlogisch.
Tabelle 8.2 zeigt die verschiedenen Arten der Adressierung, getrennt für Sender und Empfänger, die es beim MOVE Befehl gibt. Die zulässigen Arten sind mit Y gekennzeichnet.

Für jede Adressierungsart sind die Bit-Kombinationen in der Tabelle dargestellt. Sie gelten sowohl für den Fall, daß im „Register-Mode" gearbeitet wird, als auch für den „Nicht-Register-Mode". Sofern eine Adressierung ausdrücklich Register verlangt, dann wird das Bitmuster durch xxx dargestellt, da die Verschlüsselung von den wirklichen Nummern des Registers abhängig ist.

Tabelle 8.2 Zulässige Adressierungen und ihre Bitmuster

Adressierungsart	Sender	Empfänger	Bitmuster	Register
Data Reg direkt	Y	Y	000	xxx
Adress Reg direkt	Y		001	xxx
Adress Reg indirekt	Y	Y	010	xxx
Postinkrement	Y	Y	011	xxx
Predekrement	Y	Y	100	xxx
Mit Displacement	Y	Y	101	xxx
Mit Index	Y	Y	110	xxx
Absolut Kurz	Y	Y	111	000
Absolut Lang	Y	Y	111	001
PC mit Displacement				
PC mit Index				
Immediate	Y		111	100

Auf den Einwort-Befehl können noch bis zu vier weitere Worte folgen, wenn sowohl Sender als auch Empfänger absolute „lange" Adressen sind. Der längste Befehl umfaßt also fünf Worte (zehn Bytes), wogegen bei einigen Befehlen (sie betreffen die Register) ein Wort genügt.

Funktionale Gruppen

Die Befehle des MC 68000 können in acht funktionalen Gruppen zusammengefaßt werden, wenn auch in manchen Fällen die Zuordnung recht großzügig ist.

Operationen zur Daten-Bewegung:

Der Grundbefehl zum Bewegen von Daten ist MOVE. Es gibt aber, außer dieser sehr allgemeinen Instruktion, noch mehr spezielle Befehle für die Bewegung von Daten: MOVEM, bewege mehrere Register; MOVEP: transferiere Peripherie-Daten; EXG: tausche Register; LEA: lade die augenblickliche Adresse; PEA: speichere die augenblickliche Adresse; LINK: verbinde den Stack; UNLK: entkoppele den Stack; MOVEQ: bewege schnell.

Der Input- und Output-Prozeß wurde vereinfacht, da der Prozessor I/O „Memory-mapped" ist. Memory-mapped bedeutet, daß die Ein- und Ausga-

begeräte sowie die Datenströme durch Veränderung des Inhalts von Speicherstellen gesteuert werden. Das deutlichste und einfachste Beispiel eines Computers mit (wenn auch rohem) Memory-mapping ist vielleicht der Commodore 64, bei dem scheinbar die gesamte I/O-Steuerung durch PEEK und POKE erfolgt.

Ganzzahlige arithmetische Operationen:

Hier ist der große Unterschied zwischen 16-Bit Prozessoren und 8-Bit Prozessoren.
Die meisten durchschnittlichen 8-Bit Mikroprozessoren haben (Hardware-) Befehle für die Addition und Subtraktion von Bytes. Der MC 6809 ist dagegen einer der wenigen 8-Bit Prozessoren mit Maschinenbefehlen für die Multiplikation von Bytes; das Ergebnis wird dabei in einem 16-Bit Register gespeichert. Beim MC 68000 sind alle vier Grundrechenarten abgedeckt, ebenso wie die Vergleichs- und Löschbefehle. Die Additions- und Subtraktionsbefehle können alle Datentypen bearbeiten. Der Befehl MUL multipliziert zwei Worte und speichert das Ergebnis als Lang-Wort. DIV dividiert ein Lang-Wort durch ein Wort und liefert ein Wort als Ergebnis. Alle diese Befehle können sowohl Werte mit als auch ohne Vorzeichen verarbeiten.

Außerdem gibt es noch andere Instruktionen (z.B. ADDX „addiere mit Extend" oder ADDA „addiere Adressen"), die weitere Varianten der Grundbefehle sind. Sie haben jeweils das gleiche Bitmuster für den Befehlscode zu Beginn des Befehlswortes. Zur gleichen Gruppe gehören auch die beiden Testbefehle (TAS: „teste und setze" sowie TST: „teste einen Operanden".

Logische Operationen:

Die AND-, OR-, EOR- (Exclusiv Oder) und NOT-Befehle gibt es für alle Datentypen. Außerdem gibt es einen ähnlichen Satz von „immediate" Befehlen (ANDI, ORI und EORI), der logische Operationen mit Konstanten aller Größen erlaubt.

Rotations- und Schiebebefehle:

Alle normalen arithmetischen Schiebe- und Rotationsbefehle sind für alle Datentypen vorhanden. Diese Operationen können sowohl in Registern als auch im Speicher vorgenommen werden.

Operationen zur Manipulation von Bits:

Es gibt Test-, Set-, Change- und Clear- (Prüfe, Setze, Ändere und Lösche) Befehle. Die Quelle des Tests kann entweder ein Datenregister oder eine Konstante sein und die Maske kann durch fast alle Adressierungsarten angegeben werden.

Binär codierte Dezimaloperationen:

Arithmetische Operationen (mit erhöhter Geschwindigkeit) von BCD-Zahlen erhält man durch die Benutzung der Befehle: ABCD (Addiere dezimal mit Extend), SBCD (Subtrahiere dezimal mit Extend), und NBCD (Negativ dezimal mit Extend).

Operationen für die Programmsteuerung:

Die Programmsteuerung erfolgt durch einen Satz bedingter und unbedingter Verzweigungsbefehle. Übergeordnet werden sie BCC genannt. Es gibt vierzehn Bedingungen, die für eine Verzweigung getestet werden können. Zum Setzen von Bytes gibt es sechzehn Bedingungen (allgemein bezeichnet mit: SCC).
Unbedingte Verzweigungen (z.B. JMP) und Rücksprünge (z.B. RTS) vervollständigen diese Gruppe.

Operationen zur Systemkontrolle:

Systemkontrollierende Operationen erreicht man durch:
- die Verwendung der privilegierten Befehle (also Befehle, die nur im Supervisor-Modus ausgeführt werden können);
- Trap erzeugende Befehle (das sind Befehle, die die Abarbeitung von Ausnahmen veranlassen);
- Befehle, die das Status-Register benutzen oder verändern.

Folgende Ausnahmen gibt es: ein intern generierter Vorfall, wie das Ergebnis einer besonderen Instruktion oder ein interner Error; oder ein von außen kommender Vorfall, zum Beispiel ein Busfehler, ein Reset oder eine Anforderung eines externen Gerätes.

Tabelle 8.3 zeigt die wesentlichen Befehle des MC 68000 Befehlssatzes. Leichte Unstimmigkeiten zwischen diesen Mnemonics und denen eines anderen Assemblers sind dabei nicht auszuschließen.

Tabelle 8.3 Der MC 68000 Befehlssatz (alphabetisch geordnet)

Name	Beschreibung	
ABCD	decimal with extend	Dezimal-Addition mit Extend
ADD	Add	Addition
AND	Logical AND	Logisches UND
ASL	Arithmetical shift left	Arithmetische Links-Verschiebung
ASR	Arithmetical shift right	Arithmetische Rechts-Verschiebung
BCC	Branch conditionally	Bedingte Verzweigung
BCHG	Bit test and change	Maskenprüfung und Änderung
BCLR	Bit test and clear	Maskenprüfung und Löschen
BRA	Branch always	Unbedingte Verzweigung
BSET	Bit test and set	Bit testen und setzen
BSR	Branch to subroutine	Verzweigung zum Unterprogramm
BTST	Bit test	Bit testen
CHK	Check register against bounds	Register auf Grenzen prüfen
CLR	Clear operand	Lösche Operand
CMP	Compare	Vergleiche
DBCC	Test condition, decrement, branch	Bedingung prüfen, dekrementieren, verzweigen
DIVS	Signed divide	Division mit Vorzeichen
DIVU	Unsigned divide	Division ohne Vorzeichen
EOR	Exclusive Or	Exklusiv Oder
EXG	Exchange registers	Register tauschen
EXT	Sign extend	Um Vorzeichen erweitern
JMP	Jump	Sprung
JSR	Jump to subroutine	Sprung zum Unterprogramm
LEA	Load effective address	Lade effektive Adresse
LINK	Link stack	Verknüpfe Stack
LSL	Logical shift left	Logische Links-Verschiebung
LSR	Logical shift right	Logische Rechts-Verschiebung
MOVE	Move	Transferiere
MOVEM	Move multiple registers	Transferiere mehrere Register
MOVEP	Move peripheral data	Transferiere Peripherie-Daten
MULS	Signed multiply	Multiplikation mit Vorzeichen

Name	Beschreibung	
MULU	Unsigned Multiply	Multiplikation ohne Vorzeichen
NBCD	Negate decimal with extend	Dezimale Negativierung mit Extend
NEG	Negate	Negativiere
NOP	No operation	Keine Operation
NOT	One's complement	Einer-Komplement
OR	Logical Or	Logisches Oder
PEA	Push effective address	Push effektive Adresse
RESET	Reset external devices	Reset externe Geräte
ROL	Rotate left without extend	Linksrotation ohne Extend
ROR	Rotate right without extend	Rechtsrotation ohne Extend
ROXL	Rotate left with extend	Linksrotation mit Extend
ROXR	Rotate right with extend	Rechtsrotation mit Extend
RTE	Return from exception	Ausgang von Ausnahme
RTR	Return from restore	Ausgang vom Restore
RTS	Return from subroutine	Ausgang vom Unterprogramm
SBCD	Subtract decimal with extend	Dezimalsubtraktion mit Extend
SCC	Set conditional	Bedingtes Setzen
STOP	Stop	Stop
SUB	Subtract	Subtraktion
SWAP	Swap data register halves	Hälften des Daten-Registers tauschen
TAS	Test and set operand	Operand testen und setzen
TRAP	Trap	Trap
TRAPV	Trap on overflow	Trap bei Overflow
TST	Test	Test
UNLK	Unlink	Löse Verbindung

Anhang A

BASIC Vergleiche

Dieser Anhang ist eigentlich eine „Schummelei“, denn hier sind keine direkten BASIC Vergleiche.
Natürlich ist es durchaus möglich, fast jedes BASIC-Programm (besonders aber solche, die für den Sinclair Spectrum oder den ZX81 geschrieben wurden) fast Zeile für Zeile in ein SuperBASIC-Programm zu konvertieren. Ich habe aber mit dem HILO-Programm die Unsinnigkeit einer derartigen Konversion aufgezeigt (s.a. Kapitel 2).

Die meisten BASIC-Programme sind unglaublich schlecht geschrieben (außer meinen natürlich!). Ein Grund dafür, daß sie so umständlich geschrieben sind, liegt darin, daß die benutzten Programmiersprachen (z.B.: das BASIC 2 von Commodore) so primitiv sind, daß die Handbücher eigentlich alle „blau“ gefärbt sein müßten. Es gibt aber keine Entschuldigung für eine umständliche Programmierung in SuperBASIC.
Gut konstruierte Programme gibt es nicht nur in der Phantasie einiger Computer-Wissenschaftler (und ein solcher bin ich auch nicht), denn gut durchdachte Programme sind leistungsfähiger und weniger anfällig für Fehler.

Wenn jemand ein Programm in gewöhnlichem BASIC geschrieben hat, dann kann er dieses auf zwei Arten konvertieren: entweder als Feigling (der sich einbildet, SuperBASIC entspricht dem Spectrum BASIC) oder als Weiser, der versucht, es für SuperBASIC neu zu entwickeln und dafür ein effizienteres Programm erhält.

Wenn Sie ein BBC-BASIC Programm konvertieren wollen, und vorausgesetzt, daß beim Entwickeln keine Probleme entstehen, dann werden Sie feststellen, daß SuperBASIC dem BBC-BASIC weit überlegen ist.

SuperBASIC ist super!

Anhang B

Befehls-Vorrat des MC 68000

Die folgenden Tabellen wurden mit freundlicher Genehmigung von Howard W. Sams & Co. Inc. aus „Der 68000: Prinzipien und Programmierung“ von Leo J. Scanlon übernommen.

Tabelle 1 Die Kategorien der Adressierungsarten

Adressierungs-Art	Kategorie				Assembler Syntax
	Daten	Speicher	Steuerung	Variabel	
Data Register direkt	×			×	Dn
Adress-Register direkt				×	An
Register indirekt	×	×	×	×	(An)
Register indirekt mit Postinkrement	×	×		×	(An)+
Register indirekt mit Predekrement	×	×		×	−(An)
Register indirekt mit Displacement	×	×	×	×	d(An)
Register indirekt mit Index	×	×	×	×	d(An,Ri)
Absolut Kurz	×	×	×	×	xxxx
Absolut Lang	×	×	×	×	xxxxxxxx
PC relativ mit Displacement	×	×	×		d
PC relativ mit Index	×	×	×		d(Ri)
Immediate	×	×			#xxxx

Tabelle 2 68000 Befehlssatz in alphabetischer Ordnung

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen		Bedingungs-Codes
			Sender	Empfänger	X N Z V C
ABCD	ABCD Dy,Dx	8	Dn	Dn	* U * U *
	ABCD −(Ay),−(Ax)	8	−(An)	−(An)	* U * U *
ADD	ADD <ea>,Dn	8, 16, 32	All (1)	Dn	* * * * *
	ADD Dn,<ea>	8, 16, 32	Dn	Variabel	* * * * *
ADDA	ADD <ea>,An	16, 32	All	An	− − − − −
ADDI	ADDI #d,<ea>	8, 16, 32	#d	Data Variabel	* * * * *
ADDQ	ADDQ #d,<ea>	8, 16, 32	#d (2)	Variabel (1)	* * * * *
ADDX	ADDX Dy,Dx	8, 16, 32	Dn	Dn	* * * * *
	ADDX −(Ay),−(Ax)	8, 16, 32	−(An)	−(An)	* * * * *
AND	AND <ea>,Dn	8, 16, 32	Data	Dn	− * * 0 0
	AND Dn,<ea>	8, 16, 32	Dn	Variabel	− * * 0 0
ANDI	ANDI #d,<ea>	8, 16, 32	#d	Data Variabel	− * * 0 0
	ANDI #d,SR (3)	8, 16	#d	SR	* * * * *
ASL	ASL Dx,Dy	8, 16, 32	Dn (4)	Dn	* * * * *
	ASL #d,Dn	8, 16, 32	#d (5)	Dn	* * * * *
	ASL <ea>	16		Speicher Variabel	* * * * *
ASR	ASR Dx,Dy	8, 16, 32	Dn (4)	Dn	* * * * *
	ASR #d,Dn	8, 16, 32	#d (5)	Dn	* * * * *
	ASR <ea>	16		Speicher Variabel	* * * * *

Tabelle 2 (Fortsetzung)

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen: Sender	zulässige Adressierungen: Empfänger	Bedingungs-Codes X N Z V C
Bcc	Bcc <label>	8, 16	If cc, then PC + d → PC		– – – – –
BCHG	BCHG Dn,<ea> BCHG #d,<ea>	8, 32 8, 32	Dn #d	Data Variabel Data Variabel	– – * – – – – * – –
BCLR	BCLR Dn,<ea> BCLR #d,<ea>	8, 32 8, 32	Dn #d	Data Variabel Data Variabel	– – * – – – – * – –
BRA	BRA <label>	8, 16	PC + d → PC		– – – – –
BSET	BSET Dn,<ea> BSET #d,<ea>	8, 32 8, 32	Dn #d	Data Variabel Data Variabel	– – * – – – – * – –
BSR	BSR <label>	8, 16	PC → –(SP); PC + d → PC		– – – – –
BTST	BTST Dn,<ea> BTST #d,<ea>	8, 32 8, 32	Dn #d	Data, außer Konstanten Data, außer Konstanten	– – * – – – – * – –
CHK	CHK <ea>,Dn	16	If Dn < 0 or Dn > (ea), then TRAP	Data	– * U U U
CLR	CLR <ea>	8, 16, 32	Data Variabel		– 0 1 0 0
CMP	CMP <ea>,Dn	8, 16, 32	All (1)	Dn	– * * * *
CMPA	CMPA <ea>,An	16, 32	All	An	– * * * *
CMPI	CMPI #d,<ea>	8, 16, 32	#d	Data Variabel	– * * * *
CMPM	CMPM (Ay+),(Ax)+	8, 16, 32	(An)+	(An)+	– * * * *

Tabelle 2 (Fortsetzung)

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen Sender	Empfänger	Bedingungs-Codes X N Z V C
DBcc	BDcc Dn,<label>	16	If cc, then Dn − 1 → Dn; if Dn ≠ −1, then PC + d → PC		– – – – –
DIVS	DIVS <ea>,Dn	16	Data	Dn	– * * * 0
DIVU	DIVU <ea>,Dn	16	Data	Dn	– * * * 0
EOR	EOR Dn,<ea>	8, 16, 32	Dn	Data Variabel	– * * 0 0
EORI	EORI #d,<ea> EORI #d,SR (3)	8, 16, 32 8, 16	#d #d	Data Variabel SR	– * * 0 0 * * * * *
EXG	EXG Rx,Ry	32	Dn oder An	Dn oder An	– – – – –
EXT	EXT Dn	16, 32	Dn		– * * 0 0
JMP	JMP <ea>		<ea> → PC	Steuerung	– – – – –
JSR	JSR <ea>		PC → −(SP); <ea> → PC	Steuerung	– – – – –
LEA	LEA <ea>,An	32	Control	An	– – – – –
LINK	LINK An,#d	Unsized	An		– – – – –
LSL	LSL Dx,Dy LSL #d,Dn LSL <ea>	8, 16, 32 8, 16, 32 16	Dn (4) #d (5)	Dn Dn Speicher Variabel	* * * 0 * * * * 0 * * * * 0 *
LSR	LSR Dx,Dy LSR #d,Dn LSR <ea>	8, 16, 32 8, 16, 32 16	Dn (4) #d (5)	Dn Dn Speicher Variabel	* 0 * 0 * * 0 * 0 * * 0 * 0 *

Tabelle 2 (Fortsetzung)

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen		Bedingungs-Codes
			Sender	Empfänger	X N Z V C
MOVE	MOVE <ea>,<ea> MOVE <ea>,CCR MOVE <ea>,SR (6) MOVE SR,<ea> MOVE USP,An (6) MOVE An,USP (6)	8, 16, 32 16 16 16 32 32	All (1) Data Data SR USP An	Data Variabel CCR SR Data Variabel An USP	– * * 0 0 * * * * * * * * * * – – – – – – – – – – – – – – –
MOVEA	MOVEA <ea>,An	16, 32	All	An	– – – – –
MOVEM	MOVEM <list>,<ea> MOVEM <ea>,<list>	16, 32 16, 32	 Steuerung oder (An)+	Steuerung Var. oder –(An)	– – – – – – – – – –
MOVEP	MOVEP Dx,d(Ay) MOVEP d(Ay),Dx	16, 32 16, 32	Dn d(An)	d(An) Dn	– – – – – – – – – –
MOVEQ	MOVEQ #d,Dn	32	#d (7)	Dn	– * * 0 0
MULS	MULS <ea>,Dn	16	Data	Dn	– * * 0 0
MULU	MULU <ea>,Dn	16	Data	Dn	– * * 0 0
NBCD	NBCD <ea>	8		Data Variabel	* U * U *
NEG	NEG <ea>	8, 16, 32	Data Variabel		* * * * *
NEGX	NEGX <ea>	8, 16, 32	Data Variabel		* * * * *
NOP	NOP		PC + 2 – PC		– – – – –
NOT	NOT <ea>	8, 16, 32		Data Variabel	– * * 0 0

Tabelle 2 (Fortsetzung)

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen Sender	Empfänger	Bedingungs-Codes X N Z V C
OR	OR <ea>, Dn OR Dn, <ea>	8, 16, 32 8, 16, 32	Data Dn	Dn Variabel	– * * 0 0 – * * 0 0
ORI	ORI #d, <ea> ORI #d,SR (3)	8, 16, 32 8, 16	#d #d	Data Variabel SR	– * * 0 0 * * * * *
PEA	PEA <ea>	32			– – – – –
RESET (6)	RESET				– – – – –
ROL	ROL Dx,Dy ROL #d,Dn ROL <ea>	8, 16, 32 8, 16, 32 16	Dn (4) #d (5)	Dn Dn Speicher Variabel	– * * 0 * – * * 0 * – * * 0 *
ROR	ROR Dx,Dy ROR #d,Dn ROR <ea>	8, 16, 32 8, 16, 32 16	Dn (4) #d (5)	Dn Dn Speicher Variabel	– * * 0 * – * * 0 * – * * 0 *
ROXL	ROXL Dx,Dy ROXL #d,Dn ROXL <ea>	8, 16, 32 8, 16, 32 16	Dn (4) #d (5)	Dn Dn Speicher Variabel	* * * 0 * * * * 0 * * * * 0 *
ROXR	ROXR Dx,Dy ROXR #d,Dn ROXR <ea>	8, 16, 32 8, 16, 32 16	Dn (4) #d (5)	Dn Dn Speicher Variabel	* * * 0 * * * * 0 * * * * 0 *

Tabelle 2 (Fortsetzung)

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen		Bedingungs-Codes
			Sender	Empfänger	X N Z V C
RTE (6)	RTE		(SP) + → SP; (SP) + → PC		* * * * *
RTR	RTR		(SP) + CCR; (SP) + → PC		* * * * *
RTS	RTS		(SP) + → PC		– – – – –
SBCD	SBCD Dy,Dx SBCD –(Ay),–(Ax)	8 8	Dn –(An)	Dn –(An)	* U * U * * U * U *
Scc	Scc <ea>	8	If cc, then 1s → (ea); 0s → (ea)	Data Variabel	– – – – –
STOP (6)	STOP #d	16	#d → SR, then STOP		* * * * *
SUB	SUB <ea>,Dn SUB Dn,<ea>	8, 16, 32 8, 16, 32	All (1) Dn	Dn Variabel	* * * * * * * * * *
SUBA	SUBA <ea>,An	16, 32	All	An	– – – – –
SUBI	SUBI #d,<ea>	8, 16, 32	#d	Data Variabel	* * * * *
SUBQ	SUBQ #d,<ea>	8, 16, 32	#d (2)	Variabel (1)	* * * * *
SUBX	SUBX Dy,Dx SUBX –(Ay),–(Ax)	8, 16, 32 8, 16, 32	Dn –(An)	Dn –(An)	* * * * * * * * * *
SWAP	SWAP Dn	16	Dn		– – – – –
TAS	TAS <ea>	8	Data Variabel		– * * 0 0

Tabelle 2 (Fortsetzung)

Mnemonic	Assembler Syntax	Operand Größe Bits	zulässige Adressierungen		Bedingungs-Codes
			Sender	Empfänger	X N Z V C
TRAP	TRAP #<vector>		PC → −(SP); SR → −(SP); #<vector> → PC		– – – – –
TRAPV	TRAPV		If V = 1, Then TRAP		– – – – –
TST	TST <ea>	8, 16, 32	Data		– * * 0 0
UNLK	UNLK An	Unbestimmt		An	– – – – –

(1) Wenn die Operation in Bytegröße erfolgt, ist der direkte Adressregister-Mode nicht zulässig.
(2) Konstante mit einem Wert zwischen 1 und 8.
(3) Bei Operationen in Wortgröße ist der Befehl privilegiert.
(4) Source-Datenregister enthält den Schiebezähler von 0-63, wobei 0 den Wert 64 erzeugt.
(5) Data ist der Schiebezähler von 1-8.
(6) Dieser Befehl ist privilegiert.
(7) Acht Bits für Konstante, mit Zeichen auf 32 Bit erweitert.

Tabelle 3 Bedingte Prüfungen

Zusatz „cc“	Bedingung	erfüllt wenn
EQ	Gleich	$Z = 1$
Ne	Nicht gleich	$Z = 0$
MI	Minus	$N = 1$
PL	Plus	$N = 0$
*GT	Größer als	$Z \wedge (N - V) = 0$
*LT	Kleiner als	$N \veebar V = 1$
*GE	Größer oder gleich	$N \veebar V = 0$
*Le	Kleiner oder gleich	$Z \vee (N \veebar V) = 1$
Hi	Größer als	$C \wedge Z = 0$
LS	Kleiner oder gleich	$C \vee Z = 1$
CS	Carry gesetzt	$C = 1$
CC	Carry 0	$C = 0$
*VS	Overflow	$V = 1$
*VC	Kein Overflow	$V = 0$
T	immer TRUE	
F	immer FALSE	
* Zweier Komplement Arithmetik		

Symbole: $\wedge$ = Logisches UND
$\vee$ = Logisches INCLUSIV ODER
$\veebar$ = Logisches EXCLUSIV ODER

Anhang C

Bedeutung der Bits im Status-Register

Carry Flag (C): Bit 0 des Status-Registers

Das Carry Flag wird auf 1 gesetzt, wenn eine Additions-Operation zu einem „Überlauf" führt oder wenn es bei einer Subtraktion zu einem „Unterlauf" kommt. Wenn weder ein Über- noch ein Unterlauf von einer entsprechenden Instruktion verursacht wurde, dann ist das Carry-Bit 0.

Overflow Flag (V): Bit 1 des Status-Registers

Jedes Ergebnis, das die Feldgröße des Operanden überschreitet, bedeutet einen „Überlauf" und setzt das Overflow Flag auf 1.

Zero Flag (Z): Bit 2 des Status-Registers

Das Zero Flag wird gesetzt, wenn das Ergebnis einer Operation Null ist. Das kann nach einem Subtraktions- oder Dekrement-Befehl vorkommen oder wenn ein Vergleich zwischen zwei identischen Werten vorgenommen wurde. Bei jedem Ergebnis, das nicht Null ist, wird das Zero Flag zurückgesetzt. Der Zustand des Zero Flag kann geprüft werden und für bedingte Call- und Jump-Befehle benutzt werden.

Negativ Flag (N): Bit 3 des Status-Registers

Das Negativ Flag dient als Anzeiger, ob eine Zahl im Sinne der Arithmetik des Zweier-Komplements positiv oder negativ ist. Das hochwertigste Bit jedes Wertes im Zweier-Komplement wird als Positiv- oder Negativ-Anzeige benutzt, und dieses Bit wird in das Bit 3 des Status-Registers kopiert.

Extend Flag (X): Bit 4 des Status-Registers

Dieses Flag arbeitet auf die gleiche Art wie das Carry Flag, wird aber für Operationen mit größeren Zahlen benutzt, so z.B. bei Arithmetik mit erhöhter Genauigkeit.

Interrupt Maske: Bits 8 bis 10 des Status-Registers

Die Interrupt Maske besteht aus 3 Bits, die anzeigen, welche der sieben Interrupt-Ebenen augenblicklich aktiviert ist. Beachten Sie, daß die drei Masken-Bits die niederwertigsten Bits des „System"-Bytes im Status-Register sind.

Supervisor Bit (S): Bit 13 des Status-Registers

Dieses Bit wird gesetzt, um den Supervisor-Modus anzuzeigen und ist im Benutzer-Modus nicht gesetzt.

Trace Bit (T): Bit 15 des Status-Registers

Wenn dieses Bit gesetzt ist, dann zeigt es den Trace-Modus an, ansonsten ist es gelöscht.



Dieter Kiesenberg

Sinclair QL-Handbuch

1984, 121 S., kart., DM 39,80
ISBN 3-7785-1085-1

Der schon vor seinem Erscheinen mit vielen Lorbeeren bedachte QL von Sinclair will neue Maßstäbe in der Geräteklasse um 2 000 DM setzen: das vorliegende „QL-Anwenderhandbuch" soll dem Mangel an ausreichender Information über das Arbeiten mit dem QL abhelfen:
Dem Anwender beitet dieses nützliche Begleitbuch viele Hinweise und ist auch hilfreich bei der Sound- und Grafik-Programmierung.
Aus dem Inhalt:
- Die Tastur des QL
- Die Mikro-Drive-Cassette
- Bedienung der vier mitgelieferten Programme (ABACUS, ARCHIVE, EASEL und QUILL)
- Die Super BASIC-Befehle
- Das Betriebssystem QDOS
- Der Speicheraufbau des QL
- Die QL-Anschlußbelegungen
- Programmierung des 68008

Außerdem findet der Leser Programme für:
- Bildschirm-Fensteradressierung
- Catalog-Funktion
- Sound-Programmierung
- Programmierung der eingebauten Uhr

Das Buch, das auch für Einsteiger interessant sein dürfte (die Grundbegriffe bei Computern werden kurz erklärt), kann als Kauf-Entscheidung für den Interessierten dienen.

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1

Erika Hölscher

Logo auf dem Spectrum

1985, ca. 150 S., kart.,
ca. DM 35,—
ISBN 3-7785-1121-1

Am Beispiel des Sinclair-Logo wird in die Programmierung nach dem Top-Down-Prinzip eingeführt.
Abgesehen von den geringen Kenntnissen, die das Logo-Handbuch vermittelt, werden keine Grundlagen vorausgesetzt.
Logo ist ein sehr leistungsfähiges Programmierwerkzeug. Die vielfältigen Möglichkeiten dieser Sprache werden mit einfachen und dennoch interessanten Beispielen erläutert. Obwohl diese Sprache wegen ihrer Anschaulichkeit sehr für Kinder geeignet ist, findet auch der erfahrene Programmierer noch Neues. Das Sinclair-Logo unterscheidet sich nur unwesentlich von der ursprünglichen Logo-Idee. Zudem ist der ZX-Spectrum ein in seiner Klasse führender Computer bezüglich seiner Leistungsfähigkeit.

Hüthig

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1



Wolfgang Eggerichs

dBase II

Band 2: Einführung in die Programmierung mit dBASE II

1985, 191 S., zahlr. Abb., kart.,
DM 39,80
ISBN 3-7785-0987-X

Bereit erschienen: Band 1, Einführung in die Datenbanksprache dBASE II, DM 39,80
ISBN 3-7785-0986-1

In Vorbereitung: Band 3, Aufbau und Nutzung von Datenbanken mit dBASE II,
ISBN 3-7785-0988-8

Diese Buchreihe befaßt sich mit dem Datenbanksystem dBASE II, einem speziell für Mikrocomputer entwickeltem System. Dieses Datenbanksystem läuft unter den Betriebssystemen CP/M, MP/M, MS-DOS und PC-DOS.
Um den Anfänger den Einstieg in dieses doch recht mächtige Software-Werkzeug zu erleichtern, werden in den beiden ersten aufeinander abgestimmten Bänden jeweils die zu einem bestimmten Leistungsbereich gehörenden Kommandos herausgefiltert und erläutert. Zusätzlich sind kleine Aufgaben integriert, an denen der Leser theoretisch oder/ und praktisch seinen Kenntnisstand von dBASE II überprüfen kann.

Der 2. Band „Einführung in die Programmierung mit dBASE II" soll Ihnen den Schritt von der reinen Dialogarbeit zu einer verstärkten Arbeit mit den sogenannten Kommandodateien (also mit dBASE II-Programmen) zeigen. Das vorliegende Buch geht dabei besonders auf die Problemanalyse bei der Programmentwicklung ein und dementsprechend umfangreich stellt sich die dBASE II-unabhängige Erarbeitung und Aufbereitung der Programmlogik dar.

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1

Der Sinclair QL-Begleiter ist mehr als ein Handbuch für einen Microcomputer. Er hat zwei Hauptthemen: Eine gelungene Erläuterung von SuperBASIC und der diesem zugrunde liegenden Philosophie, sowie eine Beschreibung des MC 68008 Chips, das im QL verwendet wird, und des INTEL 8049 Einchip-Computers, der im QL Steuerfunktionen übernimmt.

Anhand von Programmbeispielen wird die Überlegenheit von SuperBASIC gegenüber anderen BASIC-Dialekten leicht verständlich erläutert, und der QL-Benutzer in diese BASIC-Version eingeführt. Diese Einführung schließt auch die Grafik-Programmierung mit Hilfe von SuperBASIC ein.

Die Beschreibung des 8049 Einchip-Computers von INTEL und besonders des MC 68008 Chips von MOTOROLA – einschließlich des Befehlsvorrats und der Adressierungsarten – erleichtern die volle Ausnutzung aller Möglichkeiten des Sinclair QL durch Assembler-Programmierung.